Implementación y monitorización de un sistema distribuido de inserción masiva de datos

Autor: Jorge Daniel Negrete Alvarez

Titulación: Grado en Ingeniería Informática

Especialidad: Tecnologías de la Información

Director: Guillem Corominas

Empresa: Dexma Sensors, SL

Ponente: René Serral-Gracià

Departamento: Arquitectura de Computadores (AC)

Fecha : 25/05/2015

Facultad de Informatica de Barcelona (FIB)

Universidad Politécnica de Cataluña (UPC) - BarcelonaTech

Resumen

En este mundo en que vivimos, la gran cantidad de datos que se generan y pasan por los sistemas informáticos son cada vez mayor, haciendo que las arquitecturas de los sistemas se adapten a estos cambios.

Dexma es una empresa encargada de la monitorización y análisis de datos energéticos, y cuenta con un sistema de inserción de datos por donde pasan todas las lecturas de los diferentes contadores energéticos, de agua y gas.

En este proyecto veremos la transformación de este sistema Monolítico de inserción de datos, en uno totalmente nuevo con la misión de reparar los fallos que tiene el sistema antiguo.

El objetivo del proyecto es mostrar el diseño y desarrollo de los diferentes módulos que forman el sistema distribuido orientado a los servicios, el despliegue del sistema en los entornos de trabajo y las herramientas de monitorización en tiempo real.

La falta de métricas y de suficientes herramientas para monitorizar determinaron que el sistema no esté del todo completo.

Resum

En aquest món en què vivim, la gran quantitat de dades que es generen i passen per els sistemes informàtics són cada vegada més gran, fent que les arquitectures d'aquests sistemes s'adaptin a aquests canvis.

Dexma és una empresa encarregada del monitoratge i anàlisi de dades energètiques, i compta amb un sistema d'inserció de dades per on passen totes les lectures dels diferents comptadors energètics, d'aigua i gas.

En aquest projecte veurem la transformació d'aquest sistema Monolític d'inserció de dades, en un totalment nou amb la missió de reparar els errors que té el sistema antic.

L'objectiu del projecte és mostrar el disseny i desenvolupament dels diferents mòduls que formen el sistema distribuït orientat als serveis, el desplegament del sistema en els entorns de treball i les eines de monitorització en temps real.

La manca de mètriques i de suficients eines per a monitoritzar, van determinar que el sistema no estigués del tot complet.

Abstract

In this world in which we live, the large amount of data generated and pass through the computer systems are growing, causing the architectures of these systems to adapt to these changes.

Dexma is a company responsible for the monitoring and analysis of energy data,
and it has a data entry system through which pass all readings of different energy meters, water and gas.

This project will see the transformation of this monolithic system of inserting data in entirely new one with the mission to repair the faults that have the old system.

The aim of the project is to show the design and development of different modules that form the distributed system oriented services, deployment of the system in working environments and real-time monitoring tools.

Lack of metrics and more tools to monitor, determined that the system is not entirely complete.

Índice

Capítulo 1 :Introducción

1.1 Planteamiento del Problema

La gran cantidad de información que se mueve por el mundo actualmente, provocado por la globalización y el crecimiento de internet, producen que compañías no puedan ignorar el grave problema que implica no desarrollar y adaptar su software al ritmo que imponen los negocios.

Por eso la implementación de sistemas y aplicaciones que fueron desarrolladas para correr en una plataforma en específico, deben pasar a un ambiente distribuido y heterogéneo.

Ya que las aplicaciones monolíticas, al ser software que acumulan funcionalidades bien agrupadas, tienen los siguientes fallos:

- Poca fiabilidad, debido al diseño con el cual está implementado el sistema es muy difícil hacer la replicación de componentes y esto hace que carezca de alta disponibilidad
- Mayor consumo de recursos Hardware

- Poco Escalable, replicar una funcionalidad significa replicar todas

- Complejidad para detectar fallos en el funcionamientos o en el código.

- Procesamiento de la información más lento, pero si el software tiene llamadas bloqueantes.

- Actualización de componentes más costosa.

Todos estos problemas impiden pensar en tener una entrada de mayor de datos en cualquier sistema, por es necesario el cambio de arquitectura.

1.2 Motivación

La gran cantidad de datos que hay, junto a las diferentes tecnologías que permiten el desarrollo de de sistemas distribuidos, es la motivación necesaria para poder diseñar una nueva arquitectura de un sistema.

1.3 Objetivos Generales

El objetivo principal de este proyecto es la fragmentacion de una aplicacion monolítica de insercion de datos, en modulos independientes, esto implica las siguientes tareas:

Tareas principales:

1: Diseño e implementación de distintos módulos e interfaces de conexión que conforman el proceso de inserción de datos masivos, como parte del equipo de desarrollo y de la mano del equipo de administración de sistemas.

2: Automatización de los procesos de despliegue de los diferentes modulos de la arquitectura en diferentes entornos de trabajo

3: Monitorización del sistema distribuido de inserción masiva, tanto para detección de errores(Bottlenecks, Failovers) como para medir el rendimiento del sistema.

2.1 Sistemas Distribuidos

Un sistema distribuido es una colección de computadoras independientes que aparecen ante los usuarios del sistema como una única computadora.
Cada uno de los componentes del sistema se comunican y coordinan sus acciones únicamente con el paso de mensajes.

El objetivo esencial de un sistema distribuido es proporcionar al usuario y a las aplicaciones una visión de los recursos del sistema como si estuvieran siendo gestionados por una sola máquina virtual.

Una de las características de los sistemas distribuidos es su modularidad, lo que le permite una gran flexibilidad y posibilita su escalabilidad, definida como la capacidad del sistema para crecer sin aumentar su complejidad ni disminuir su rendimiento. Uno de los objetivos del diseño de un sistema distribuido es extender la escalabilidad a la integración de servicios.

Ventajas respecto a un sistema centralizado

- **Escalabilidad**: consecuencia de su modularidad
- **Disponibilidad**: mediante replicación de recursos
- **Tolerancia a Fallos:** la replicación permite esto
- **Paralelismo:** entre los modulos del sistema, mejora el rendimiento.

Desventajas respecto a un sistema centralizado

- **Consistencia:** mantener que el estado global del sistema siga igual incluso si un módulo se haya caído
- **Problema de red:** la red de interconexión entre los modulos puede ser una fuente de problemas para el sistema.
- **Seguridad:** Al ser cada módulo del sistema una aplicacion que presta un servicio provoca más puntos en los cuales se pueda sufrir un ataque desde fuera

2.2 Big Data

Básicamente se refiere a la enorme cantidad de datos que desde hace unos años se genera constantemente a partir de cualquier actividad que implique el uso de la informática de alguna forma.

Implementar una solución alrededor de Big Data implica la integración de diversos componentes y proyectos que en conjunto forman el ecosistema necesario para analizar grandes cantidades de datos.

Es decir hacer uso de herramientas que están construidas para ser ejecutadas en sistemas distribuidos a gran escala diseñados para tratar con grandes volúmenes de información, analizando tanto datos estructurados como no estructurados.

2.3 Bottle, UWsgi , Nginx

Bottle, Uwsgi y Nginx son herramientas utilizadas para levantar pequeñas y sencillas aplicaciones web en python.

Bottle es un wsgi(web server gateway) web framework, bastante liviano y simple, utilizado para codificar pequeñas apps web. Las apps hechas en Bottle son single thread server, asi que no son aconsejables subirlas a producción ya que generaría un cuello de botella importante.

Para resolver este problema aparece Uwsgi. Es una aplicación encargada de levantar, configurar y mantener aplicaciones web, como en nuestro caso hechas en bottle. Es bastante eficiente y liviano dado que está programado en c.
En sistemas que tienen un gran número de peticiones, es aconsejable tener un Nginx por delante de la aplicacion uwsgi, para asi tener una mayor contención de la carga de datos.
Nginx es un servidor web bastante liviano y sencillo, que este caso haría de proxy redireccionando todas las peticiones hacia la aplicacion web levantada con uwsgi.
Nginx nos permite esto ya que viene con un plugin que nos permite trabajar con el protocolo uwsgi.

2.4 RabbitMQ

Rabbitmq es un servidor de intercambio de mensajes, con su sistema de colas almacena los mensajes que llegan de los emisores, los almacena a la espera de que sean consumidos por los receptores. Rabbit entra en la categoría de middleware de mensajería.
El proyecto RabbitMQ consta de diferentes partes:

- El servidor de intercambio RabbitMQ en sí mismo
- Pasarelas para los protocolos HTTP, XMPP y STOMP.
- El plugin Shovel (pala) que se encarga de copiar (replicar) mensajes desde un corredor de mensajes a otros

2.5 Apache Storm

Apache Storm es un sistema distribuido de computación en realtime, libre y de código abierto.Realiza de manera fácil y eficiente el procesado en tiempo real de flujo de datos.

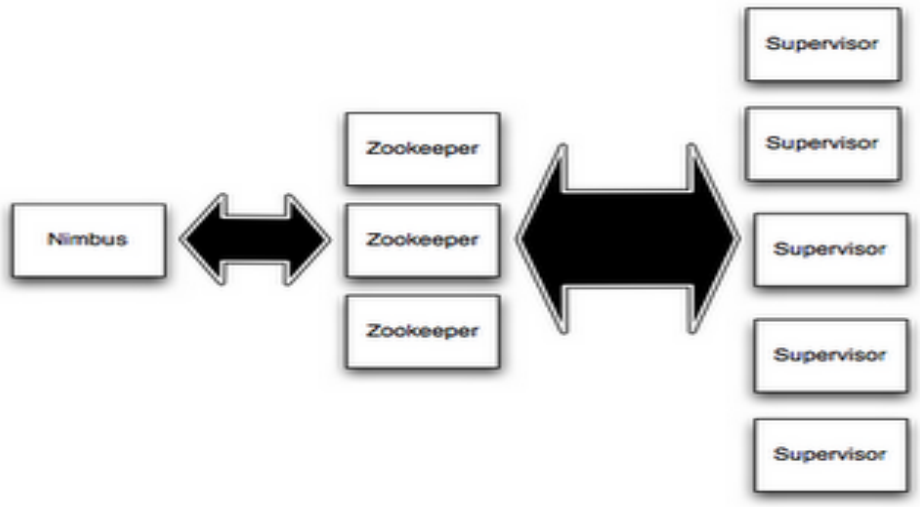
Storm es en real time lo que hadoop es en procesos batch.Es rápido, escalable, tiene tolerancia a fallos, garantiza que tus datos serán procesados, es facil de levantar y operar.

Conceptos Principales

Para entender mejor cómo trabaja Storm es importante entender los siguientes conceptos principales.

Storm Cluster

Storm Cluster es donde se levanta una topología storm. Hay dos clases de nodos que forman parte de un Cluster, el master y los workers.
El nodo master levanta un daemon llamado nimbus que es el encargado de distribuir el trabajo entre los workers y controlar el estado de ellos por si hay algún fallo.
Los nodos worker corren un demonio llamado Supervisor. Este demonio escucha el trabajo que le asigna el nimbus y levanta los procesos correspondientes según el trabajo que se le ha asignado. Cada worker ejecuta parte de una topología. Una topología entera consiste en muchos workers trabajando a través de muchas maquinas



Y finalmente toda la coordinación que hay entre el nimbus y los supervisors está en manos del Zookeeper. El nimbus y los supervisors son stateless, es decir todo el estado de cada uno de estos demonios los guarda el Zookeeper. Esto significa que se puede reiniciar el nimbus o los supervisors y estos volverán como si nada hubiera pasado.

Topologia

La lógica de una aplicacion en tiempo real está empaquetado en una topología storm. Una topología storm es un grafo de Spouts y Bolts conectados entre y si van pasando un stream de datos. Estos conceptos están explicados más abajo.

Spouts

Los spouts son la fuente donde se generan los streams de datos. Generalmente estos datos son recogidos de una fuente externa (Rabbit), y el spout es el encargado de recogerlos y meterlos a la topologia.

Bolts

Todas las tareas que se realizan en un topología son programadas dentro de los bolts. Los bolts pueden realizar funciones, filtrar datos, comunicarse con un servicio externo (Base de datos, api, etc) y mas.

Los streams que entran por un bolt son procesados y remitidos a un siguiente bolt con una estructura diferente. Cada vez que un bolt termina de procesar un Stream este llama a la función ack para decir que ha terminado de procesar un mensaje al sistema de control.

Tareas

Todas las tareas son ejecutadas por los spots y bolts dentro de una topología. Cada tarea corresponde a un thread de ejecución.Según el tiempo que tarda esta tarea se puede aumentar el número de threads que la realicen.

Workers

Topologías se ejecutan a través de uno o varios procesos workers. Cada proceso worker es una máquina virtual Java y ejecuta un grupo de todas las tareas dentro de la topología. Si hay 300 tareas en paralelo dentro de la topología, cada worker ejecutará 6 tareas(6 threads, thread por tarea). Storm trata de repartir equitativamente la carga de trabajo entre los workers.

2.6 MongoDB

MongoDB es una Base de Datos NoSql de tipo documental que nos provee un alto rendimiento, alta disponibilidad y escalado automático.
Mongo es de tipo documental, lo que quiere decir que la estructura de los datos que almacena esta formada por campos que son clave y valor(JSON).Estos valores pueden ser otros documentos o un array o un array de documento.

Las ventajas de usar documentos son:

- Los documentos corresponde a un tipo de datos nativo utilizado en muchos lenguajes de programación
- Reduce las joins en las lecturas.

Mongo es una base de datos con alto rendimiento en las lecturas y escrituras, aparte con los índices las lecturas son aún más rápidas .

2.7 Redis DB

Redis es un motor de base de datos de tipo clave/valor,esto quiere decir que El modelo de datos de Redis se basa en la estructura de datos del tipo diccionario o tabla de hashes que relaciona una llave a un contenido almacenado en un índice. La principal diferencia entre Redis y otros sistemas similares es que los valores no están limitados a ser de tipo string, otros tipos de datos están soportados:

- Listas (Lists) de strings
- Sets de strings (colecciones de elementos desordenados no repetidos)
- hashes donde la llave y el valor son del tipo string

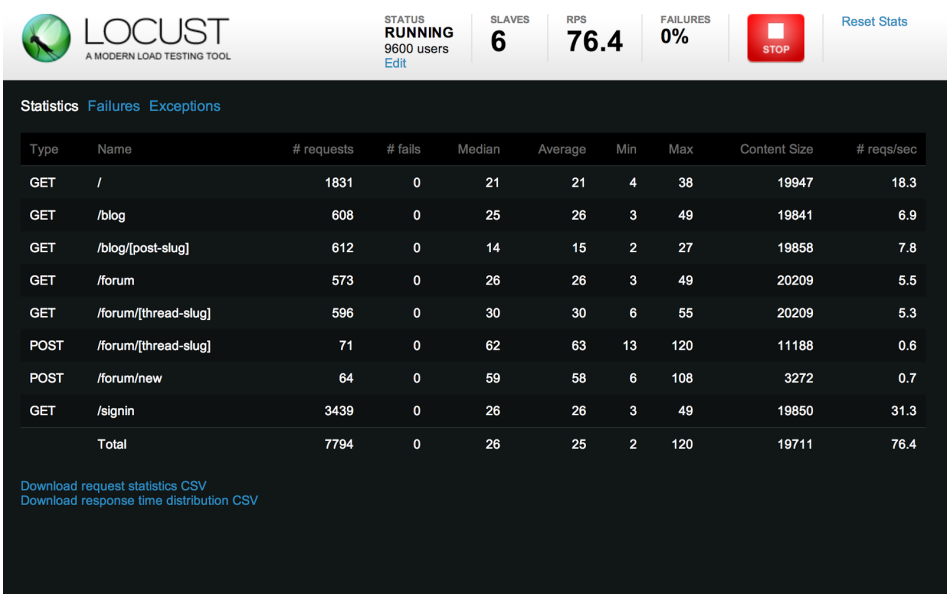
Redis normalmente guarda la información en la memoria RAM.Se puede hacer que los datos sean persistentes de 2 formas, una es hacer snapshots (capturas), aunque no seria realmente durable ya que estos son asincrónicos al transferir la memoria al disco cada cierto tiempo. Desde la versión 1.1 la mejor alternativa es usar un archivo del tipo *appendonly* gracias a un sistema de Journaling el cual escribe en este archivo cada modificación que se realice sobre los datos en memoria pudiendo regenerar los datos. Esto genera un costo en el rendimiento, pero se puede configurar de 2 formas, *always*, que escribiera cualquier cambio en el instante, o *everysec* que lo hará segundo tras segundo.

2.8. Locust

Locust es una herramienta distribuida encargada de realizar test de carga sobre aplicaciones para poder averiguar cuántos usuarios de manera concurrente pueden atacar la aplicacion. Normalmente utilizado para realizar pruebas sobre aplicaciones web.

Locust es una herramienta hecha en python, así que los escenarios deben estar programados en el lenguaje python. El interaccion de cada usuario lanzado por locust responde a lo programado en el escenario.

Todos los usuarios son monitorizados por la herramienta, que nos da una interface web en la cual nos muestra las estadísticas de los test, con los resultados finales.



Locust Interfaz Web

2.9 Jenkins

Jenkins es un software de Integración continua open source escrito en Java. Está basado en el proyecto Hudson y es, dependiendo de la visión, un fork del proyecto o simplemente un cambio de nombre.

Jenkins proporciona integración continua para el desarrollo de software. Es un sistema corriendo en un servidor que es un contenedor de servlets, como Apache Tomcat. Soporta herramientas de control de versiones como CVS, Subversion, Git, Mercurial, Perforce y Clearcase y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como scripts de shell y programas batch de Windows. El desarrollador principal es Kohsuke Kawaguchi. Liberado bajo licencia MIT, Jenkins es software libre.

2.10 Ansible

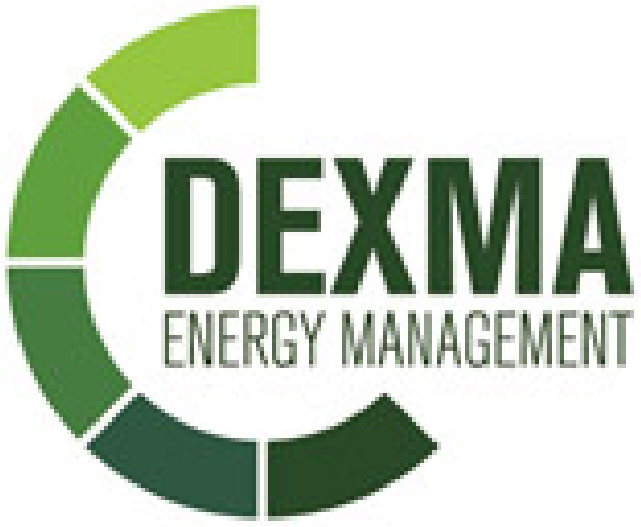
Ansible es una plataforma de software libre para configurar y administrar computadoras. Combina instalación multi-nodo, ejecuciones de tareas ad hoc y administración de configuraciones. Adicionalmente, Ansible es categorizado como una herramienta de

orquestación.[1] Maneja nodos a través de SSH y no requiere ningún software remoto adicional (excepto Python 2.4 o posterior para instalarlo. Dispone de módulos que trabajan sobre JSON y la salida estándar puede ser escrita en cualquier lenguaje. Nativamente utiliza YAML para describir configuraciones reusables de los sistemas.

2.11 Dexma

DEXMA es una empresa que se dedica a la recolección, monitorización y análisis de consumos energéticos en tiempo real.

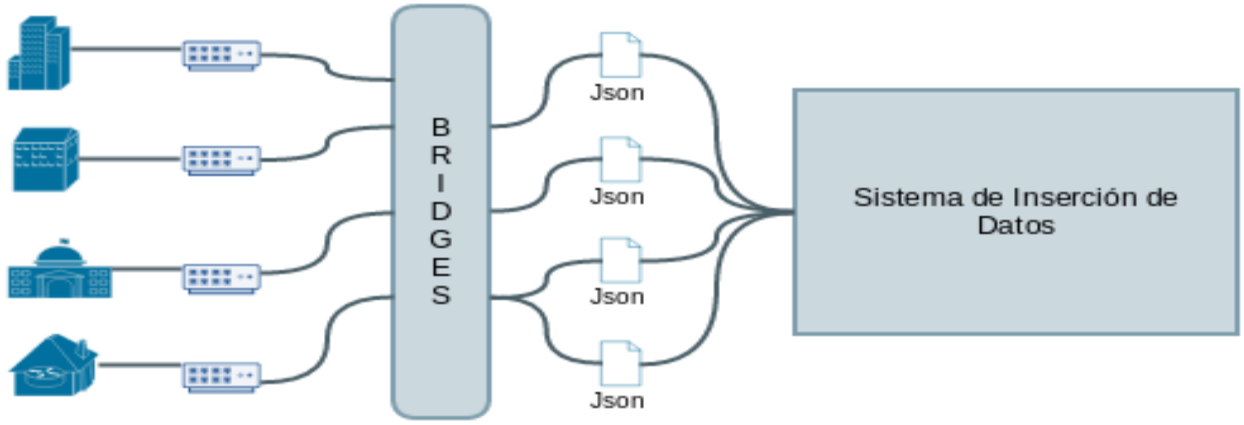
Miles de medidores envían millones de datos energéticos (eléctricos, gas, agua o temperatura, etc) a una plataforma online, que los preprocesa y los almacena en base de datos.



Logo Dexma

Envio de datos

La manera en que dexma de recibir los diferentes datos de los medidores energéticos es:



Envio de datos

Los diferentes medidores energéticos van conectados a un tipo de concentrador. Este concentrador gestiona las lecturas y envía los datos a la plataforma de inserción. Todos estos datos pasan por los modulos bridges, que son procesos encargados de obtener los mensajes, y darle un formato estándar de entrada hacia el sistema de inserción de datos.

DEXCell Energy Manager

DEXCell es una plataforma web, en la cual los clientes de la empresa pueden ver los datos de los diferentes medidores energéticos que tienen enviando lecturas hacia el sistema de inserción.

Antes de empezar a enviar lecturas, todo cliente debe registrar primero el tipo de concentrador de donde se obtienen las lecturas. Un concentrador es un hardware encargado de gestionar los datos de todos los medidores energéticos conectados a él. Existen varios tipos de concentradores, y según

cual envía los datos de muchas formas.

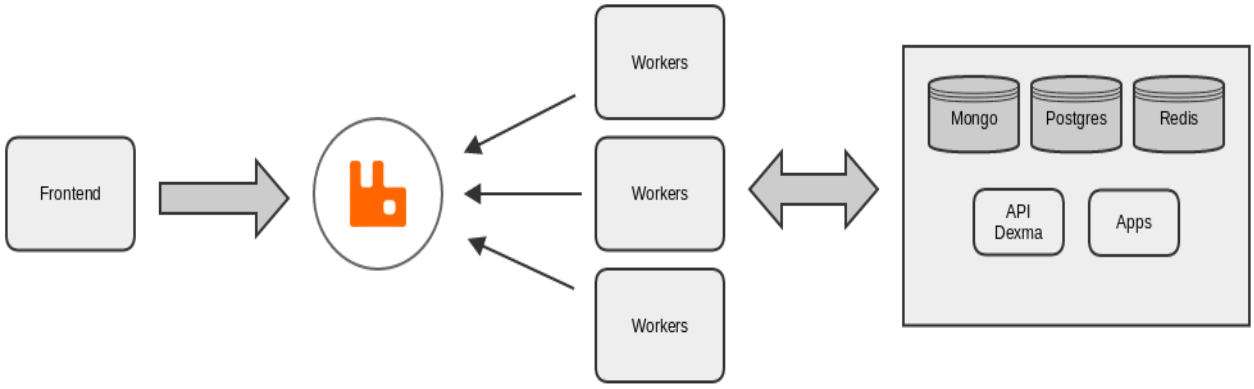
Al concentrador registrado en dexcell está identificado por un número (mac) único en la plataforma web.

Sistema de Alertas

El sistema de alertas es una herramientas que ofrece el DEXCell en la cual permite al cliente configurar alarmas, que avisen notificaciones si el consumo energético de un contador en concreto es mayor a un valor máximo(el cliente lo configura al crear la alerta)

3.Punto de partida

El nuevo sistema de inserción de datos se empezó a construir teniendo como referencia las funcionalidades del antiguo sistema de inserción.



Anterior arquitectura del sistema de inserción

Como podemos observar la antigua arquitectura del sistema de inserción estaba basada en el modelo colas-workers, en la cual los datos de entrada eran almacenados en un sistema de colas(RabbitMQ) y luego eran procesados por un número de workers.

Cada worker al procesar un mensaje realizaba lo siguiente:

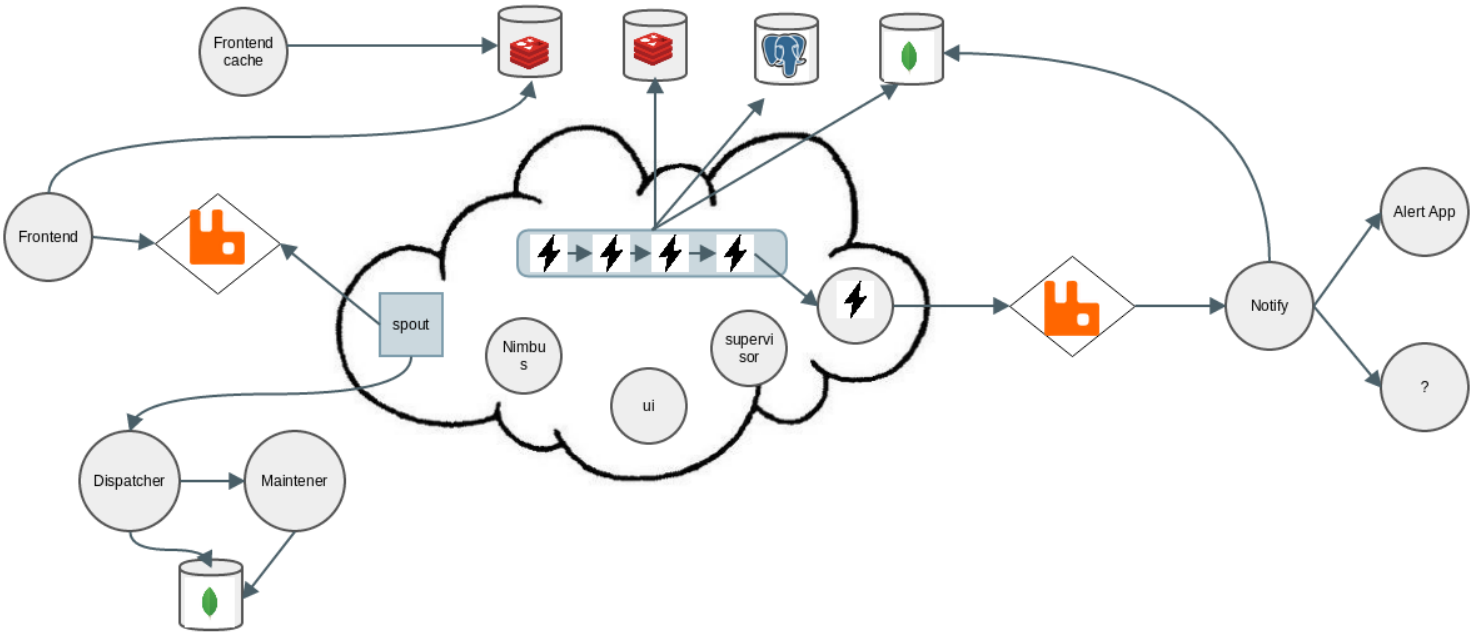
- Conversión y validación de los datos
- Conexión a diferentes servicios externos (Mongodb,Postgresql,Redis, Apis)
- Gestion y distribucion de Colas(RabbitMQ)
- Tratamiento de Alertas(Herramienta Dexcell)

Viendo la gran cantidad de tareas que realiza cada worker, mezclado con la complejidad de algunas de ellas y el tiempo que tarda en realizar peticiones a servicios externos, nos a llevado a pensar si este sistema es el adecuado para procesar un gran número de mensajes.

Por eso, partiendo de este software, teniendo en cuenta lo que hace y los inconvenientes que tiene, se empezó a diseñar un nuevo sistema de inserción de datos que haga todo lo que hacía el anterior, más el añadido de corregir los inconvenientes mencionados anteriormente.

Teniendo como punto de partida el antigua sistema de inserción, se diseñó una nueva arquitectura totalmente distribuida, teniendo como módulo central una topología hecha en storm, acompañada de diversos modulos en python.

El diseño del nuevo sistema de inserción es el siguiente:



Nueva Arquitectura del Sistema de Inserción

Como podemos observar en la imagen, la nueva arquitectura cuenta con un mayor número de modulos producto de la división de las funcionalidades del anterior sistema.

Como novedad podemos apreciar que el módulo Frontend anterior a sido dividido en dos modulos, Frontend y Frontend Cache. Tambien aparecen dos nuevos modulos que son el Dispatcher y el Maintainer.

Como parte central tenemos una instancia de una topología Storm, donde podemos apreciar los diferentes componentes que la forma.

Y para finalizar, aparece el sistema de alertas, que antes formaba parte del módulo encargado de la inserción de datos(Workers) y en la nueva arquitectura se ha externalizado, transformándose en un módulo totalmente independiente.

4.1 Modulo Frontend

El módulo frontend es la puerta de entrada de los datos al sistema de inserción, tanto en la antigua arquitectura como en la nueva, su función es la de recibir los datos de los contadores y publicarlos a las colas del RabbitMQ.

Anteriormente el Frontend era una aplicación rest hecha en bottle, que recibía los datos a traves de la llamada http://insert.dexcell.com/insert-json.htm en la cual la informacion estaba en el body de la trama http. El mensaje llegaba estructurado en un documento json, en donde venía toda la información de la lectura o lecturas del contador energético.

Con el nuevo Frontend, se ha heredado gran parte del codigo del antiguo, y se le han aplicado los siguientes cambios:

-Seguridad y Autenticación por Token

En el nuevo Frontend, se pedirá como parámetro en la llamada rest el token que identifica el concentrador que está mandando este mensaje. Con esto descartamos con mayor rapidez aquellos mensajes que no son lecturas de contadores registrados en el DEXCell.

Ademas de recibir el token como parametro se pide tambien el id del concentrador que está haciendo la llamada. Tanto el id, como el token son parámetros únicos, dados al usuario en el momento de crear el concentrador en la aplicacion Dexcell.

La nueva llamada quedaria asi:

https://host-url/readings?source_key=<Id>&x-dexcell-source-token=<Token>

-Pool de conexiones con el RabbitMQ

Con cada llamada entrante se establece una conexión con el Rabbit para publicar los mensajes en la cola correspondiente. En el anterior frontend no existía un control sobre las conexiones establecidas con el rabbit y a medida que llegan un mayor número de llamadas, al mismo momento, se genera un número de conexiones mayor que el sistema operativo puede soportar, así esto generaba la pérdida de datos, debido a falta de recursos del sistema.

El nuevo frontend cuenta con un pool de conexiones, esto quiere decir que como máximo el módulo generará 'N' número de conexiones consecutivas hacia el rabbit. Así tendremos un control sobre los recursos generados por el módulo y evitaremos errores por falta de ellos.

Para poder generar un pool de conexiones se cambió la librería encargada de crear y gestionar las conexiones al Rabbit. En el frontend antiguo estaba la librería Pika, bastante utilizada para establecer conexiones con el rabbit, pero en la misma documentación lo dice que "en procesos multithread no es posible utilizar la misma instancia de conexión en todo los threads".

La librería Kombu es la utilizada en el nuevo frontend para establecer comunicación con el Rabbit. Kombu es una libreria mas completa que Pika, nos brinda muchas más funcionalidades y una de ellas es un pool de conexiones.

-Escalabilidad

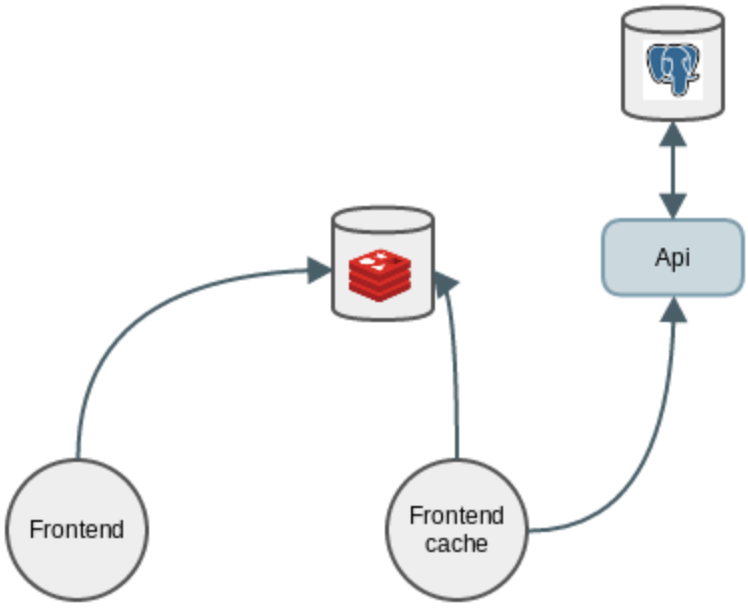
Cada vez que un mensaje entra, el frontend debe obtener información del concentrador que envía los datos, para poder validar los parámetros de autenticación y también saber a que cola del Rabbit debe publicar la información.

Esto implicaba que por cada mensaje , el frontend debe conectarse a base de datos(Postgresql) y obtener la información del concentrador. El tiempo que implicaba esta acción era muy elevado, así que se puso un redis por delante para que haga el trabajo de memoria caché.

Por eso anteriormente el frontend realizaba dos tareas concurrentes, una era la de procesar los mensajes de entrada y la segunda era la de conseguir y actualizar la información de todos los concentradores que estaban registrados en el DEXCell en el Redis.

Al tener un módulo con dos tareas, imposibilitaba el poder hacer replicación de este componente, así que se separaron estas funcionalidades como dos modulos independientes, uno encargado de procesar mensajes(Frontend) y otro de actualizar la información en el redis(Frontend_cache).

4.2 Modulo Fronted Cache



Flujo de trabajo Frontend Cache

El módulo frontend cache es el encargado de obtener y actualizar el Redis con información de los diferentes concentradores registrados en el DEXCell.

Como ya se habia dicho anteriormente este módulo formaba parte de una funcionalidad del módulo Frontend.

No obstante al nuevo módulo tiene algunas diferencias respecto a la anterior funcionalidad.

La primer diferencia es que la información de los diferentes concentradores se la obtiene realizando una llamada a la API Rest interna que tiene la empresa, a diferencia de conectarse directamente a la base de datos Postgres. Con esto se quiere lograr un mayor control de las conexiones a la base de datos haciendo que otra entidad se encargue de gestionar esto.

Luego otra diferencia está en los datos almacenados en el redis, en ellos estan los tokens de los diferentes concentradores para poder verificar la autenticación en el momento de recibir la llamada, que antes no lo hacía.

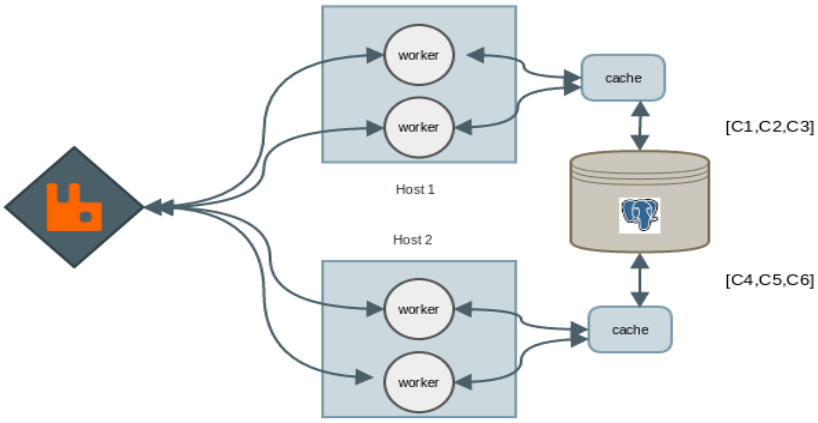
Con este módulo conseguimos quitarle complejidad al código y comportamiento del Frontend así como la posibilidad de poder replicarlo.

4.3 Gestion de Colas

Anteriormente,en el sistema de inserción antiguo, la distribución de colas entre los workers se realizaba de la siguiente manera:

En el RabbitMQ habia un numero limitado de colas(20) en donde se almacenaba la información de todos los clientes de la empresa, esto quiere decir que una sola cola recibía datos de diferentes clientes.

La información de todas estas colas estaba almacenada en una base de datos postgres.



Antiguo flujo de selección de colas

El proceso encargado de levantar los workers dentro de un host traía de la base de datos postgres la información de un grupo de colas del Rabbit y la guardaba dentro de una estructura de datos en memoria. Así cada worker dentro de una misma máquina, llamaba un método interno que cogía una cola de la estructura de datos y la entregaba para que empiece a ser consumida. Cada worker en un host distinto consumía de un rango de colas distinto, ya que una cola solo podía estar siendo consumida por un solo worker a la vez.

Este funcionamiento tenias muchas deficiencias, como ser:

- Retrasos en la inserción de datos, porque al tener una cola almacenando información de diferentes clientes, podía darse el caso que un cliente envíe más datos que otros con los que comparte la cola, esto provocaba que un worker este mayor tiempo procesando mensajes de un solo cliente, retrasando la inserción de datos de los demás
- Si un proceso moria, habia colas que estarian sin ser consumidas por un tiempo hasta que el proceso sea levantado de nuevo
- Dificultar al escalar el sistema, porque al levantar workers en un nuevo host era necesario reagrupar las colas en el postgres y resetear las cache de los demás procesos.

Por eso en el nuevo sistema de inserción se ha externalizado esta funcionalidad, creando un subsistema formado por dos modulos, llamados Dispatcher y Maintainer, encargados de realizar la gestión y distribucion de las colas.

4.3.1 Modulo Dispatcher

El módulo dispatcher, es un micro servicio, encargado del control y distribucion de las colas del rabbit, basándose en el estado de las colas y de estrategias de entrega.

A través de una Api Rest el dispatcher dice a los consumidores(Spout) que cola deben consumir en cada momento, ademas de otras funcionalidades.

Estados de las colas

La entrega de una cola u otra en cada momento dependerá del estado en que se encuentren.

Los estados de las colas son:

Idle : La cola no está siendo consumida por nadie

Worker: La cola está siendo consumida actualmente

Empty : La cola no tiene mensajes para consumir

Delete: La cola a sido eliminada

Pause: La cola está bloqueada

El estado de las colas está almacenado en una colección de la base de datos mongodb, ya que tanto el dispatcher, como el módulo maintainer(Explicado más adelante) son los encargados de modificar el estado de las colas.

API Rest

El dispatcher ofrece sus servicios a través de una api rest implementada en bottle.

Con estas llamadas el dispatcher distribuye las colas entre los distintos consumidores, ademas de modificar el estado de cada una de ellas.

Los servicios que ofrece el dispatcher son:

Entrega de colas:

Con la llamada **/queue/next** el dispatcher entrega la siguiente cola a ser consumida, teniendo en cuenta la estrategia de entrega que se esta utilizando. El dispatcher va al mongo y pide una cola que esté no esté siendo consumida en ese momento(IDLE), ademas de otros criterios que lo determina la estrategia.

Algo a tener en cuenta es que esta llamada no dará la misma cola a dos consumidores diferentes, ya que una cola solo puede estar siendo consumida por un worker a la vez.

La cola al ser entregada pasa a estar en estado WORKER.

Devolver cola:

Toda cola luego de ser consumida debe ser devuelta, para que así pueda ser entregada a otro consumidor. Con la llama **/queue/<id>/release** el consumidor devuelve la cola al dispatcher, pasando como parámetro por la url, el identificador de la cola en cuestión. Con esta llamada, esa cola pasa a de estar en estado WORKER a estado IDLE.

La entrega de cola también se puede realizar al momento de pedir una nueva cola, pasando como parámetro en el body de la llamada http el identificador de la cola.

Bloquear cola:

Con la llamada /queue/<id>/pause, el dispatcher bloqueara una cola para que no sea consumida por ningún worker, el identificador de la cola viene como parámetro de entrada en la url de la llamada. El dispatcher cambiará el estado de la cola a Pause en la base de datos.

Desbloquear cola:

Con la llamada /queue/<id>/unpause, el dispatcher desbloqueara una cola que estaba anteriormente bloqueada, así podrá ser dada a los consumidores para que su contenido sea procesado.

Parar Dispatcher:

Con la llamada /stop el dispatcher deja de ofrecer su servicios a los consumidores(Entregar Colas), así sin la necesidad de matar el proceso el dispatcher deja de entregar colas, solo se permite devolverlas.

Arrancar Dispatcher:

Con la llamada /start el dispatcher puede volver a entregar colas como siempre a los consumidores. Levanta el servicio previamente parado.

Ver Estatus:

Con la llamada /queue/status el dispatcher devuelve el estado de todas las colas almacenadas en base de datos.

Estrategias de Entrega

Las estrategias de entrega son criterios que deciden que cola se debe entregar en cada momento. Según el estado en que están y la estrategia con la que está trabajando el dispatcher, la cola entregada puede ser diferente en un momento específico. Las dos estrategias con las que actualmente está trabajando el dispatcher son:

FIFO

Esta estrategia es la más común y simple que hay en todo sistema de gestión de trabajo. Con esta estrategia el dispatcher dará todas la colas a los consumidores sin tener en cuenta el número de mensajes almacenado .

El criterio que tomara el dispatcher es:

- que no esté siendo consumida
- que lleve el mayor tiempo sin ser entregada a un consumidor

Lo malo de esta estrategia es el tiempo que se tarda en entregar colas con mensajes.

Ya que depende del número de colas que hay con mensajes, a mayor número de colas sin mensajes, el tiempo de procesado del sistema subirá.

LRU

En esta estrategia a diferencia de la FIFO se tendrá en cuenta tres cosas:

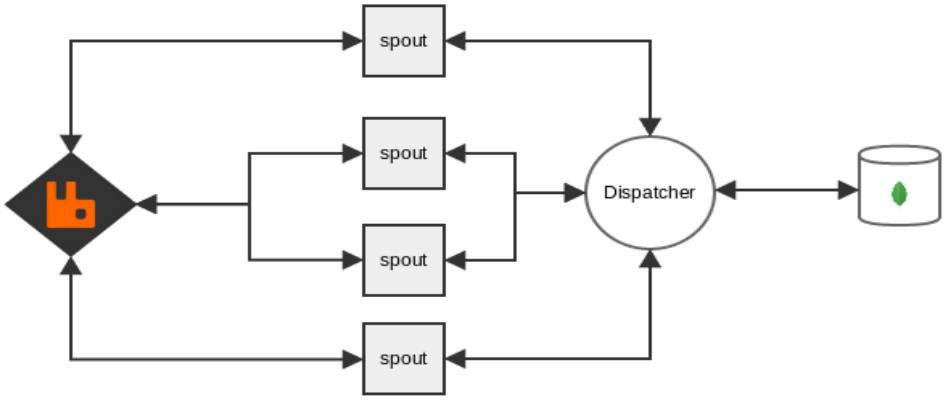
- que tenga mensajes
- que no esta bloqueada
- que lleve más tiempo sin ser consumida

Así siguiendo este orden el dispatcher decide que colas dar y el rendimiento del sistema subirá

Otras posibles estrategias

De momento solo hay dos, pero en el futuro se está pensando programar más estrategias como ser entregar la cola más prioritaria o con más mensajes en un momento dado.

Interaccion con los Spouts



Interaccion Spouts,Dispatcher y Rabbit

Los spouts, elementos de la topología storm, son los encargados de consumir datos de las colas del rabbit , e introducirlos dentro de la topología para que sean procesados.

Los spouts carecen de información acerca de las colas del rabbit por eso necesitan preguntarle al dispatcher sobre que cola deben consumir. La interacción entre los spouts y el dispatcher es la siguiente:

- Los spouts hacen la llamada http://dispatcher-host-url/queue/next pidiendo por una cola para consumir.
- El dispatcher recibe la llamada y va a la base de datos,obtiene el identificador de una cola según la estrategia que está usando y retorna el identificador en la respuesta http.
- El spout recibe el identificador y empieza a consumir mensajes de la cola hasta haber consumido un número N de mensajes o hasta dejarla totalmente vacía.
- Y finalmente el spout pide una nueva cola usando la llamada anterior, solo que esta vez pasa como parámetro el identificador de la cola anteriormente consumida, ademas de notificar si la cola esta vacia.

Ventajas

Con esta nueva infraestructura de tener un módulo encargado de gestionar las colas del rabbit tenemos las siguientes ventajas respecto al anterior:

- Escalar el número de spouts sin necesidad de tocar el dispatcher.
- Poder programar estrategias de entrega de las cola.
- Mayor control del estado de las colas.
- La pérdida o fallo de algún consumidor, no afecta a la entrega de las colas.

Desventajas

Aunque sea muy pocas aquí hay algunas desventajas de tener el dispatcher gestionando las colas:

- El tiempo de entrega dependerá mucho del número de consumidores que están pidiendo colas, ya que todas las peticiones tienen un punto final en común que es

la base de datos y el dispatcher hace llamadas de lectura y escrituras atómicas sobre esta.

- Si el dispatcher cae, el sistema de inserción deja de funcionar
- Controlar los fallos de red y la comunicación, entre los spouts y el dispatcher.

4.3.2 Modulo Maintainer

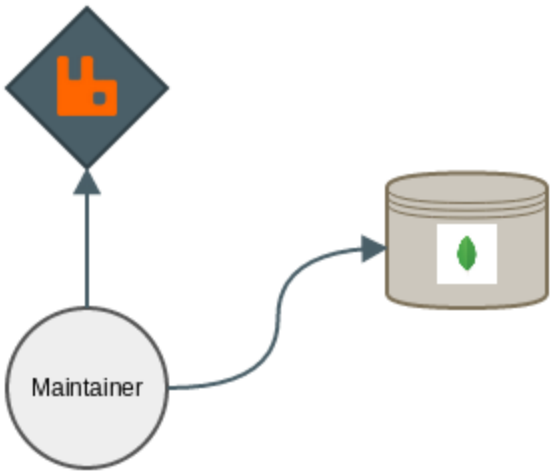
Módulo encargado de gestionar toda la información acerca de las colas del Rabbit. Realizando ciertas tareas mantiene actualizada la información de las colas en la Base de datos MongoDB.

Anteriormente esta función no existía ya que los datos eran introducidos manualmente a una base de datos, al igual que la creacion de colas en el rabbitmq.

Esto era posible porque el número de colas era pequeño(20), y estas colas recibían información de diferentes cuentas de usuarios.

La desventaja de esto era que si una cuenta enviaba una gran cantidad de datos a una cola, esto provocaba un retraso en la inserción de datos de otras cuentas que enviaban menos datos.

Por eso en el nuevo diseño cada cuenta tendrá su propia cola, que será creada y gestionada por el maintainer.



Modulo Maintainer

Tareas

Crear y actualizar información de las colas:

En esta tarea el maintainer pregunta, haciendo uso de la Api interna de la empresa, por todas los identificadores de las cuentas que existen y su estado(Activado o Desactivado). Luego con cada identificador verifica si existe la información de la cola en el mongodb. En caso que no, la crea y crea la cola en el rabbit asignando como id el identificador de la cuenta. Si una cuenta está desactivada o eliminada se activa el flag de eliminada, en la información de la cola en el mongo si esta cola existe.

Actualizar colas Vacias

En esta tarea el maintainer pregunta al Rabbit, a través de su API, información de todas las colas con el número de mensajes que tiene y actualizar la información de cada cola diciendo si tienen mensaje o no. Esto ayuda mucho al dispatcher a entregar las colas.

Controlar Timeouts

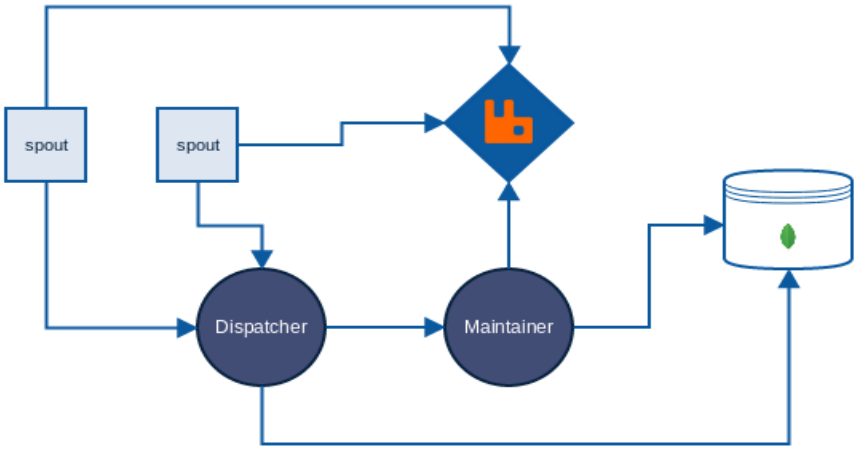
Puede darse el caso que una cola no sea devuelta nunca, es decir si un consumidor pide una cola, pero por razones desconocidas reinicia su estado o se cae y vuelve a levantarse, esa no será devuelta y nunca volverá a ser entregada por el dispatcher.

Para evitar eso, el maintainer controla el tiempo que una cola puede estar siendo consumida. El tiempo por defecto es una hora, ya que sería imposible que un consumidor se quede procesando la información de una cola por más de una hora.

Eliminar Colas

En esta tareas el maintainer busca toda la colas que tenga el flag de eliminada activado y elimina la información de la cola en la base de datos, ademas de eliminar la cola de RabbitMQ.

Interaccion Dispatcher-Maintainer



Flujo Final Dispatcher-Maintainer

El comportamiento del dispatcher depende mucho del trabajo del maintainer, ya que cada una de las tareas afecta a la estrategia que está utilizando el dispatcher.

La estrategia LRU del dispatcher depende mucho de algunas tareas que realiza el maintainer para conseguir información sobre las colas, una de ellas es saber que colas tienen mensajes y cuáles no.

Por esta razón el dispatcher está constantemente verificando cual es el estado del maintainer, si esta vivo o no enviando mensajes y esperando una respuesta.

En caso que no haya respuesta del maintainer, el dispatcher cambia la estrategia en la que esta por la FIFO, ya que toda la información que hay sobre las colas está desactualizada.

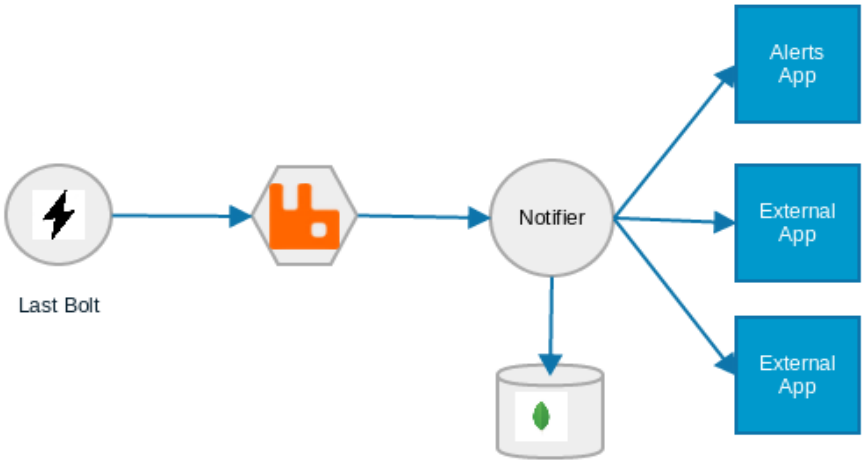
4.4 Notificador de Alertas

Anteriormente las notificación de alertas era la última tarea que realizaba un worker al procesar un mensaje, y consistía en ver si el concentrador al que pertenecía el mensaje tenía una alerta configurada en la aplicacion DEXCell.

En caso de no tener una alerta configurada, el procesado del mensaje terminaba ahí y se continuaba con uno nuevo de la cola. Pero si tenia una alerta configurada se procedía a:

- Ver si el valor del mensaje es mayor al límite configurado en la alerta
- En caso de que si, avisar al cliente por medio de mail, mensajes sms, etc.

Esta funcionalidad era muy penalizadora a nivel de tiempo, y perjudicaba bastante la velocidad de inserción en general , así que se decidió separar el codigo de toda esa funcionalidad y diseñar un sistema externo a lo que es la inserción de datos, que vaya procesando las alertas a su ritmo sin perjudicar en el tiempo de la inserción.



Nuevo Flujo Sistema De Alertas

En el nuevo sistema de inserción, la parte de procesar las alertas se ha externalizado, creando un aplicacion totalmente independiente a la topología Storm.

El código se ha heredado totalmente del antiguo sistema y se ha creado una aplicacion externa. Y a esto se le ha añadido una arquitectura nueva por delante, encargada de comunicar la salida de la topología storm y la aplicacion de alertas.

En esta nueva arquitectura sobresalen dos componentes nuevos que son el servidor RabbitMQ y el Notificador de alertas

El rabbitMQ se encargará de almacenar en una cola todas las notificaciones de alertas publicados por el último bolt de la topología storm.

El “Notificador de Alertas” es un proceso encargado de recoger las notificaciones de las colas y entregarlas a la aplicaciones de alertas las cuales se encargaran de procesarlas.

En el mongodb se guarda toda la información respecto a la configuración de las alertas, los aplicaciones(endpoints) a la cuales deben ser mandadas las notificaciones de las alertas y datos acerca de la conexión hacia la aplicacion(URL,Metodos Http,etc).

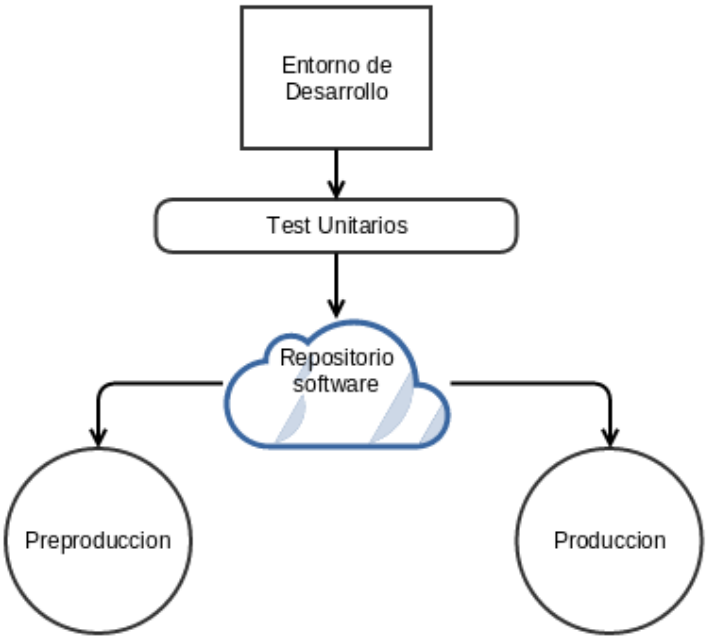
Actualmente solo existe la aplicación de alertas que se ha heredado del anterior sistema de inserción, pero se realizó este diseño pensando en que en el futuro haya más aplicaciones.

5.Entornos de Trabajo

En este apartado hablaremos de las fases por la que cada uno de los modulos desarrollados han tenido que pasar hasta llegar finalmente a la puesta en marcha en en un entorno de Producción al servicio del cliente.

Las fases son las siguientes:

- Entorno de Desarrollo
- Test Unitarios
- Repositorio Software
- Entorno Preproducción
- Entorno Producción



Entornos de trabajo

Entorno De Desarrollo

Etapa en la cual abarca desde el comienzo del ciclo de vida del módulo, hasta la obtención de una primera versión del código. En esta fase se lleva a cabo la implementación del los modulos.

Test Unitarios

Test utilizados para tener un cierto grado de seguridad de saber que lo que se ha implementado funcione. Cada módulo del sistema cuenta con su set de test unitarios, que son lanzados cada vez que se realiza un cambio en el código y antes de subir al repositorio software.

La librería utilizada para realizar los test en los modulos python se llama pytest.

Repositorio Software

Luego de que el módulo logre pasar su set de test unitarios, es almacenado en un repositorio software, esperando el momento de ser desplegado en el siguiente entorno.

El repositorio de Software utilizado en el desarrollo del sistema es subversión.

Entorno de Preproducción

En el entorno de preproducción se lleva a cabo las pruebas finales de la aplicación antes de su paso final al entorno de producción, donde se pondrá en funcionamiento en un escenario real.

Estas pruebas de integración entre los diferentes modulos que forman el sistema son:

- Funcionales y estructurales
- De rendimiento
- De tolerancia a fallos
- De seguridad

Si estas pruebas, conocidas como de aceptación, resultan satisfactorias, entonces la aplicación está ya lista para su paso al entorno de producción.

Entorno de Producción

El entorno de producción contiene en todo momento la versión activa de la aplicación. Los usuarios finales tienen acceso a la aplicación implantada en este entorno.

La aplicación se despliega en el entorno de producción procedente de la versión existente en el entorno de preproducción.

6.Despliegue del Sistema

En esta apartado hablaremos de la manera en la cual desplegamos todos los modulos almacenados en el repositorio, en los entornos de preproducción y producción.

6.1 Despliegue de los modulos

Los diferentes modulos, hechos en python, estan estructurados de la misma forma para realizar el despliegue sobre los entornos de producción y preproducción.

Esta estructura de despliegue está dividida en tres partes:

- Codigo python
- Configuracion Uwsgi
- Servidor web Nginx

Codigo Python

Es el código que se encuentra almacenado en el repositorio software(Subversion).

En cada despliegue, se descarga el código perteneciente a la última versión subida al repositorio software.

Configuracion Uwsgi

Uwsgi es el servicio elegido para poder iniciar las aplicaciones, ya que no solo se encarga de levantar y construir la aplicacion web, si no que también se encarga de supervisar el estado de la aplicacion en todo momento.

Partiendo del punto de que ya tenemos instalado de antemano esta herramienta en el entorno de trabajo, es necesario editar un fichero(.ini o .xml), en donde se especifica todos los parámetros necesarios para que esta herramienta construya la aplicacion.

Es necesario tener los ficheros de cada aplicacion en una direccion conocida por el servicio para que sea ejecutado.

Los ficheros de configuración Uwsgi tienen la siguiente estructura:

```
[uwsgi]
plugin = python
virtualenv = /opt/venvs/is3frontend
pythonpath = /opt/apps/is3frontend
wsgi-file = /opt/apps/is3frontend/server.py
module = insertserver
single-interpreter = true
buffer-size = 10000
socket = 127.0.0.1:3031
listen = 128
master = true
workers = 8
threads = 1
worker-reload-mercy = 8
exit-on-reload = true
no-orphans = true
stats = 127.0.0.1:9191
vacuum = true
logto = /var/log/uwsgi/%n.log
```

El siguiente fichero de configuración corresponde a la aplicacion frontend, todas las demas aplicaciones siguen la misma configuracion.

Entre los parametros a destacar tenemos:

plugin = Indica el lenguaje en que está codificado la aplicacion web

virtualenv = Indica el entorno donde estan instaladas las dependencias de la aplicacion

uwsgi-file = Indica el path del fichero donde se encuentra la aplicacion que debe levantar

Buffer Size = Indica el tamaño máximo que debe tener las peticiones de entrada

socket = indica el puerto donde estara escuchando la aplicacion

listen = Número máximo de conexiones consecutivas permitidas

workers = Número de workers que tendrá la aplicacion trabajando en paralelo

master = con esto decimos que la aplicacion sera un solo proceso master que se encargará de gestionar los workers.

worker-reload-mercy = Tiempo que tienen los worker para finalizar una petición antes de morir.

vacuum = Indicamos que en caso de reiniciar la aplicacion, todas las conexiones abiertas deben ser cerradas.

Actualmente Uwsgi entiende los protocolos Http,FastCGI, CGI y finalmente uno llamado Uwsgi. El protocolo con mejor rendimiento es el Uwsgi, así que la aplicacion estara esperando llamadas Uwsgi.

Las peticiones que se recibirán son Http, así que es necesario poner una capa más por delante, encargada de realizar la traducción del protocolo Http a Uwsgi.

Servidor web Nginx

El servidor web nginx será el encargado de recibir las peticiones http, transformarlas al protocolo uwsgi y redireccionarlas hacia el socket abierto por la aplicacion web.

Un ejemplo del fichero de configuración en el servidor nginx es el siguiente:

```
server {
   listen   80 default_server ;
   server_name  is3.dexcell.com ;
   location / {
      include uwsgi_params;
      uwsgi_pass 0.0.0.0:3031;
      uwsgi_read_timeout 180;
   }
   error_page 500 502 503 504 /50x.html;
   location = /50x.html{
      root /usr/share/nginx/html;
   }
```

}

En el siguiente fichero podemos observar como todas las peticiones que llegaran por el puerto 80 son redirigidas por el puerto 3031 con otro protocolo, en este caso uwsgi.
Aparte de esto, tener un servidor web como nginx delante de la aplicacion uwsgi, también nos permite contener un número mayor de llamadas concurrentes.

6.2 Automatización

Jenkins y Ansible son las Herramientas utilizadas para automatizar todos los pasos para el despliegue de los modulos del sistema en los entorno de preproduccion y produccion.

La manera que llevamos a cabo el despliegue de los modulos es la siguiente:

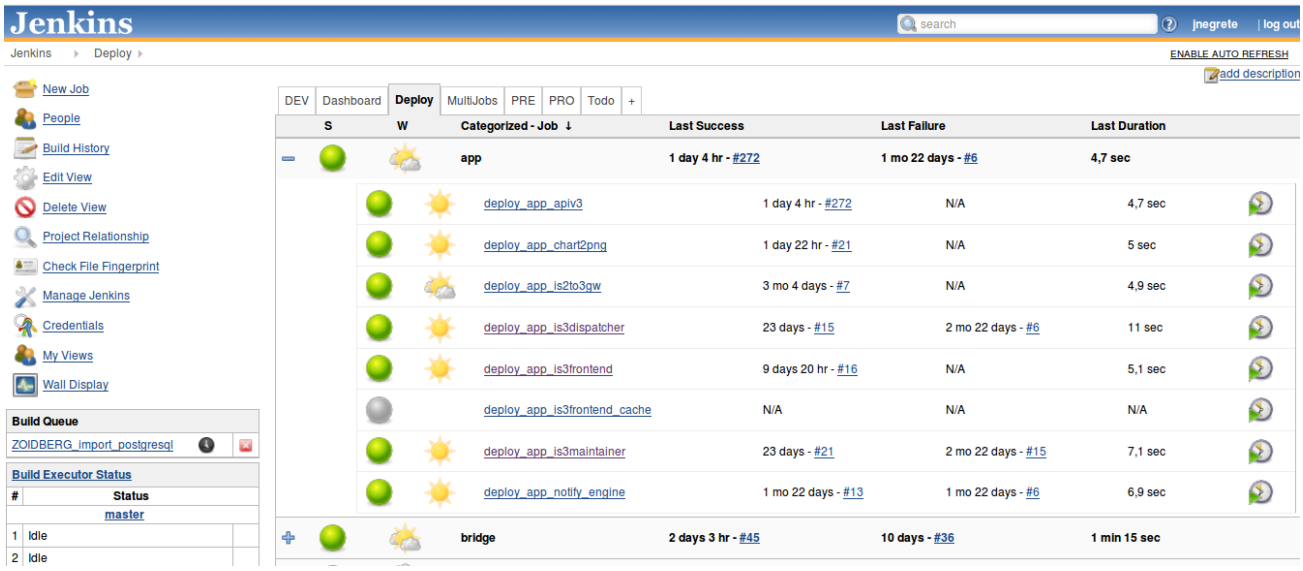
1. Haciendo uso de Ansible, generamos varios scripts(uno por cada módulo), encargados de realizar las siguientes tareas:

- Conectarse a la máquina donde funcionara el módulo del sistema
- Descargar el código del repositorio software donde se encuentra
- Instalar las librerías necesarias para el funcionamiento del código
- Reiniciar el servicio Uwsgi que ejecuta el código de la aplicacion.

Los scripts de ansible se encuentran almacenados en una máquina remota y de la ejecución de estos se hará cargo el sistema de integración continua, jenkins.

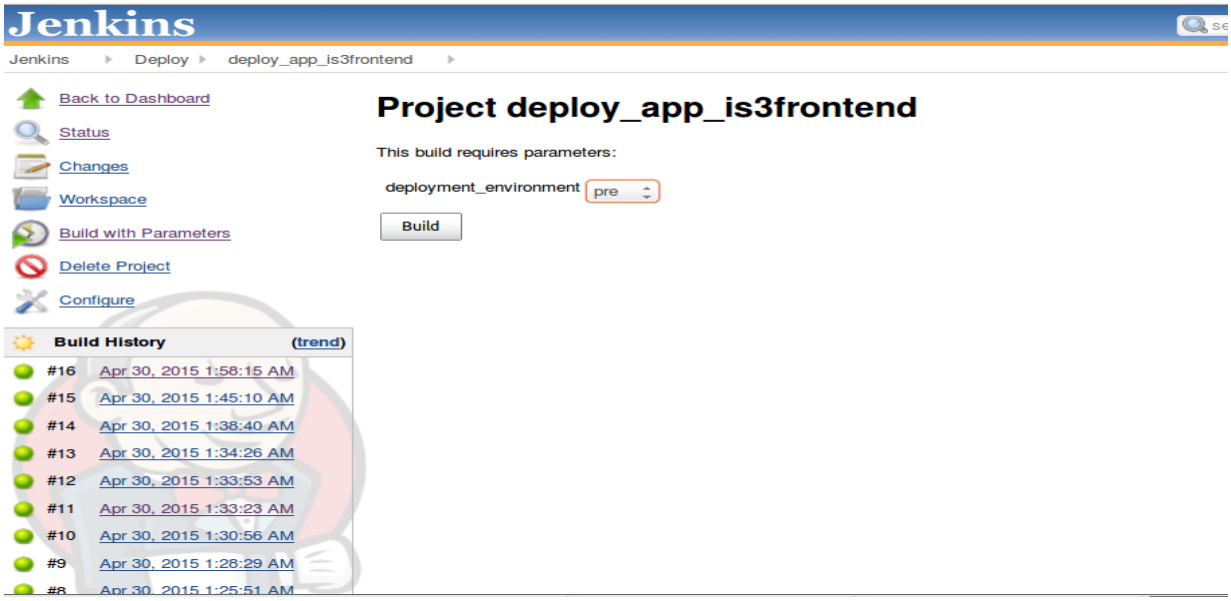
2. La aplicacion web del jenkins corre en un servidor web apache tomcat que está en la mism máquina donde se encuentran los scripts de Ansible.

Para evitar tener que ejecutar los scripts de ansible manualmente, hacemos uso de Jenkins y su interfaz web para crear una tareas que hagan este trabajo automáticamente.



Tareas del Jenkins

Al seleccionar unas de las tareas para realizar el despliegue, es obligatorio seleccionar el entorno donde se desea hacer el despliegue, si es preproducción o producción.

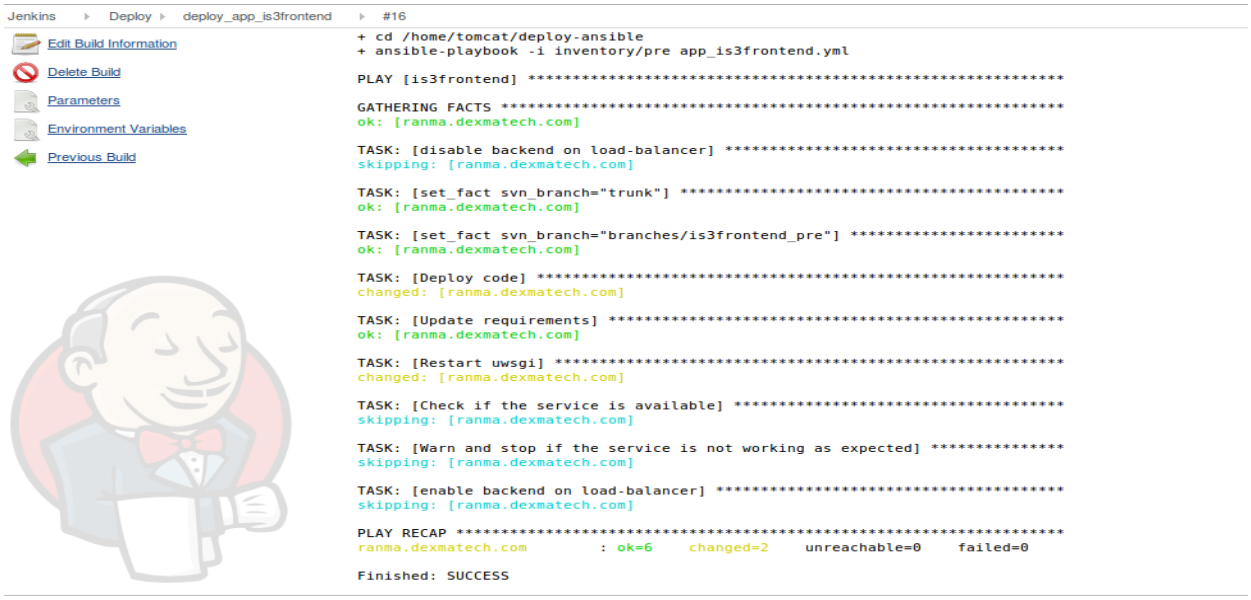


Despliegue Modulo Frontend

Luego de seleccionar este parámetro, se realiza la tarea que se encargará de ejecutar el script Ansible, que recibiendo como parámetro el entorno(Preproduccion y

Produccion), ejecutara todo el despliegue y todos los pasos mencionados con anterioridad.

Jenkins nos ofrece una terminal online, en la cual podremos ver todos los pasos que realiza la tarea y verificar si el resultado ha sido exitoso o si ha ocurrido algún tipo de error.



Consola Online del Jenkins

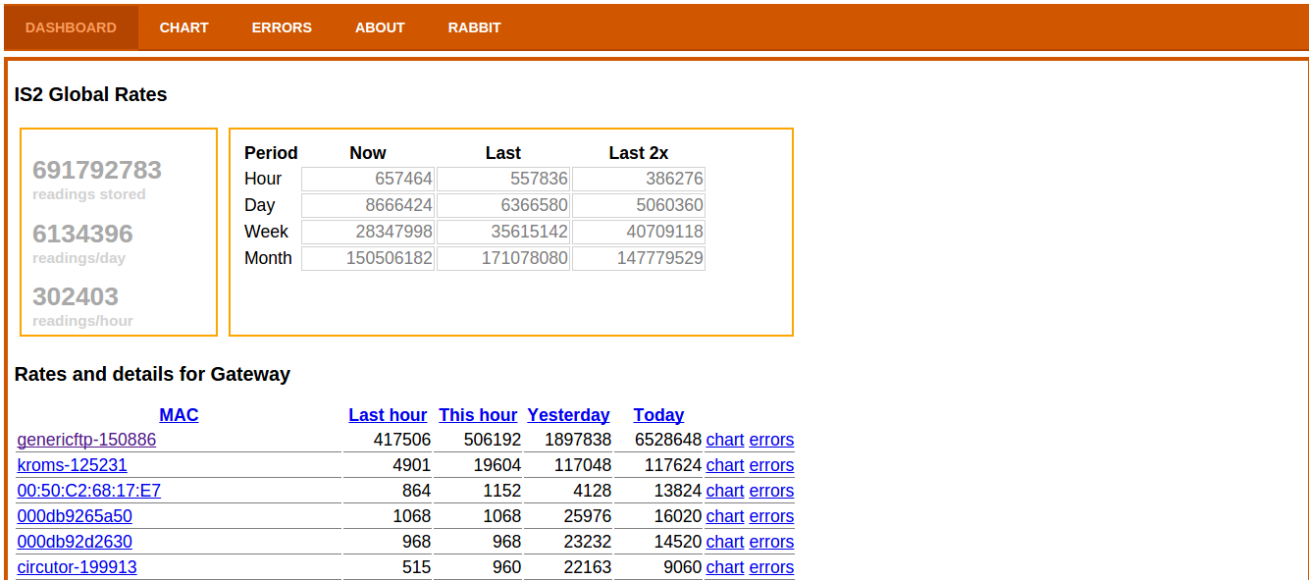
7.Herramienta de Monitorización

Las herramientas de monitorización son aquellas que nos ayudan a entender el comportamiento del sistema, así como poder detectar posibles anomalías que puedan ocurrir en alguna parte de la arquitectura del sistema.

De momento se cuenta con tres interfaces web, que nos brindan información sobre tres partes diferentes del sistema.

- Administrador del Frontend
- Interfaz Web RabbitMQ

7.1 Administrador del Frontend

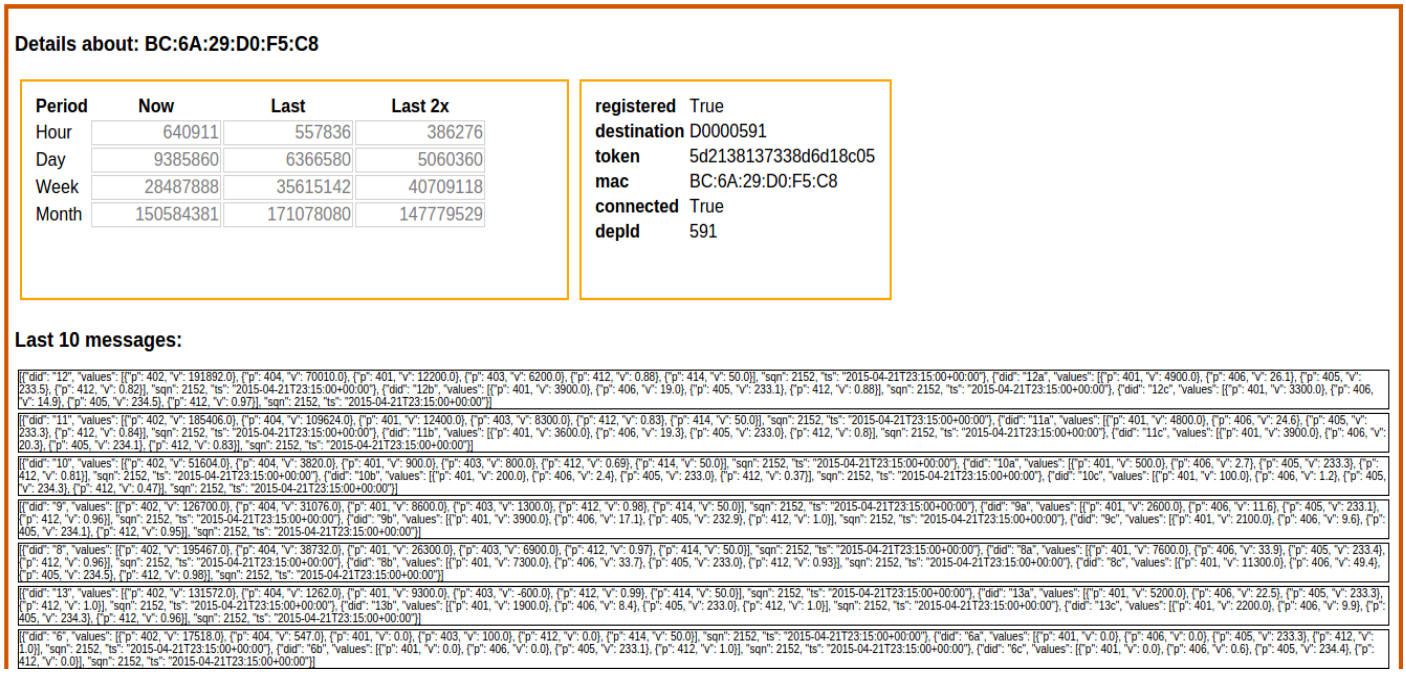


Pagina Principal Admin Frontend

El administrador del frontend es una interface web que se creó para monitorizar los mensajes que pasan por este módulo.

En la página principal podemos observar estadísticas sobre el número de mensajes que el frontend ha procesado en total, por horas, por semanas, por meses.

También podemos ver una tabla con todos los concentradores que envían datos al sistema. En esta tabla veremos estadisticas del numero de mensajes que llegan clasificados por concentrador, ademas de un enlace a otra pagina para poder ver información más detallada de cada concentrador.



Details about: BC:6A:29:D0:F5:C8

Period	Now	Last	Last 2x
Hour	640911	557836	386276
Day	9385860	6366580	5060360
Week	28487888	35615142	40709118
Month	150584381	171078080	147779529

registered	True
destination	D0000591
token	5d2138137338d6d18c05
mac	BC:6A:29:D0:F5:C8
connected	True
depId	591

Last 10 messages:

[{"did": "12", "values": [{"p": 402, "v": 191892.0}, {"p": 404, "v": 70010.0}, {"p": 401, "v": 12200.0}, {"p": 403, "v": 6200.0}, {"p": 412, "v": 0.88}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "12a", "values": [{"p": 401, "v": 4900.0}, {"p": 406, "v": 26.1}, {"p": 405, "v": 233.5}, {"p": 412, "v": 0.82}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "12b", "values": [{"p": 401, "v": 3900.0}, {"p": 406, "v": 19.0}, {"p": 405, "v": 233.1}, {"p": 412, "v": 0.88}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "12c", "values": [{"p": 401, "v": 3300.0}, {"p": 406, "v": 14.9}, {"p": 405, "v": 234.5}, {"p": 412, "v": 0.97}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "11", "values": [{"p": 402, "v": 185406.0}, {"p": 404, "v": 109624.0}, {"p": 401, "v": 12400.0}, {"p": 403, "v": 8300.0}, {"p": 412, "v": 0.83}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "11a", "values": [{"p": 401, "v": 4800.0}, {"p": 406, "v": 24.6}, {"p": 405, "v": 233.3}, {"p": 412, "v": 0.84}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "11b", "values": [{"p": 401, "v": 3600.0}, {"p": 406, "v": 19.3}, {"p": 405, "v": 233.0}, {"p": 412, "v": 0.8}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "11c", "values": [{"p": 401, "v": 3900.0}, {"p": 406, "v": 20.3}, {"p": 405, "v": 234.1}, {"p": 412, "v": 0.83}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "10", "values": [{"p": 402, "v": 51604.0}, {"p": 404, "v": 3820.0}, {"p": 401, "v": 900.0}, {"p": 403, "v": 800.0}, {"p": 412, "v": 0.69}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "10a", "values": [{"p": 401, "v": 500.0}, {"p": 406, "v": 2.7}, {"p": 405, "v": 233.3}, {"p": 412, "v": 0.81}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "10b", "values": [{"p": 401, "v": 200.0}, {"p": 406, "v": 2.4}, {"p": 405, "v": 233.0}, {"p": 412, "v": 0.37}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "10c", "values": [{"p": 401, "v": 100.0}, {"p": 406, "v": 1.2}, {"p": 405, "v": 234.3}, {"p": 412, "v": 0.47}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "9", "values": [{"p": 402, "v": 126700.0}, {"p": 404, "v": 31076.0}, {"p": 401, "v": 8600.0}, {"p": 403, "v": 1300.0}, {"p": 412, "v": 0.98}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "9a", "values": [{"p": 401, "v": 2600.0}, {"p": 406, "v": 11.6}, {"p": 405, "v": 233.1}, {"p": 412, "v": 0.96}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "9b", "values": [{"p": 401, "v": 3900.0}, {"p": 406, "v": 17.1}, {"p": 405, "v": 232.9}, {"p": 412, "v": 1.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "9c", "values": [{"p": 401, "v": 2100.0}, {"p": 406, "v": 9.6}, {"p": 405, "v": 234.1}, {"p": 412, "v": 0.95}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "8", "values": [{"p": 402, "v": 195467.0}, {"p": 404, "v": 38732.0}, {"p": 401, "v": 26300.0}, {"p": 403, "v": 6900.0}, {"p": 412, "v": 0.97}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "8a", "values": [{"p": 401, "v": 7600.0}, {"p": 406, "v": 33.9}, {"p": 405, "v": 233.4}, {"p": 412, "v": 0.96}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "8b", "values": [{"p": 401, "v": 7300.0}, {"p": 406, "v": 33.7}, {"p": 405, "v": 233.0}, {"p": 412, "v": 0.93}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "8c", "values": [{"p": 401, "v": 11300.0}, {"p": 406, "v": 49.4}, {"p": 405, "v": 234.5}, {"p": 412, "v": 0.98}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "13", "values": [{"p": 402, "v": 131572.0}, {"p": 404, "v": 1262.0}, {"p": 401, "v": 9300.0}, {"p": 403, "v": -600.0}, {"p": 412, "v": 0.99}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "13a", "values": [{"p": 401, "v": 5200.0}, {"p": 406, "v": 22.5}, {"p": 405, "v": 233.3}, {"p": 412, "v": 1.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "13b", "values": [{"p": 401, "v": 1900.0}, {"p": 406, "v": 8.4}, {"p": 405, "v": 233.0}, {"p": 412, "v": 1.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "13c", "values": [{"p": 401, "v": 2200.0}, {"p": 406, "v": 9.9}, {"p": 405, "v": 234.3}, {"p": 412, "v": 0.96}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

[{"did": "6", "values": [{"p": 402, "v": 17518.0}, {"p": 404, "v": 547.0}, {"p": 401, "v": 0.0}, {"p": 403, "v": 100.0}, {"p": 412, "v": 0.0}, {"p": 414, "v": 50.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "6a", "values": [{"p": 401, "v": 0.0}, {"p": 406, "v": 0.0}, {"p": 405, "v": 233.3}, {"p": 412, "v": 1.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "6b", "values": [{"p": 401, "v": 0.0}, {"p": 406, "v": 0.0}, {"p": 405, "v": 233.1}, {"p": 412, "v": 1.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}, {"did": "6c", "values": [{"p": 401, "v": 0.0}, {"p": 406, "v": 0.6}, {"p": 405, "v": 234.4}, {"p": 412, "v": 0.0}], "sqn": 2152, "ts": "2015-04-21T23:15:00+00:00"}]

Información de un concentrador en concreto

En esta página podemos observar información detallada sobre un concentrador en concreto, como ser información general, id, token, cuenta a la que pertenece.

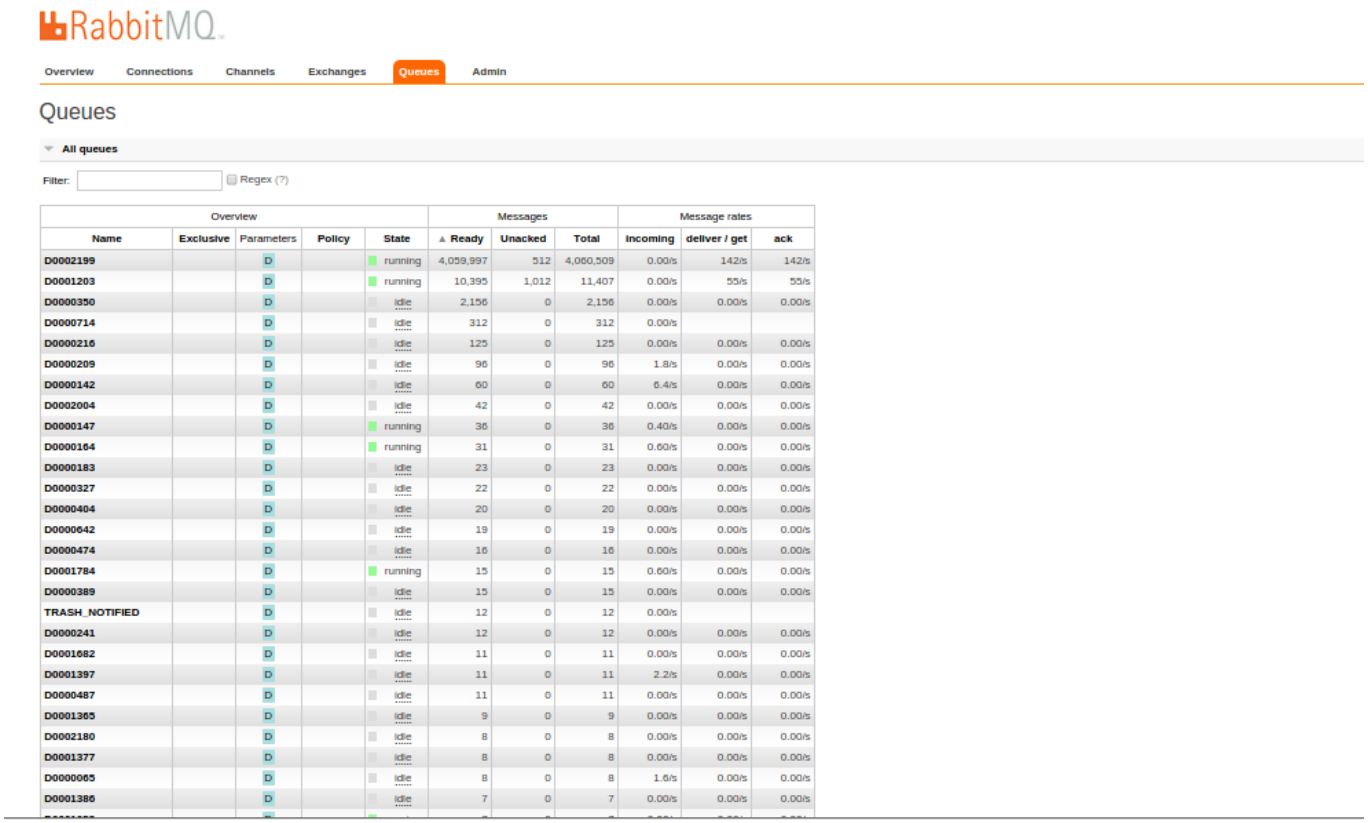
Ademas de mostrar los 10 ultimos mensajes que pasan por este módulo.

7.2 Interfaz Web RabbitMQ

El servidor rabbitmq nos ofrece una aplicacion web, en el cual podemos monitorizar todas las colas existentes, así como el número de mensajes que contienen cada una de ellas.

Esta herramienta nos ayuda a verificar muchas cosas, como ser la velocidad de publicación y consumo que hay, el número de colas que hay creadas(Deben coincidir

con el número de cuentas que existen) o el número total de mensajes que hay almacenados en el servidor.



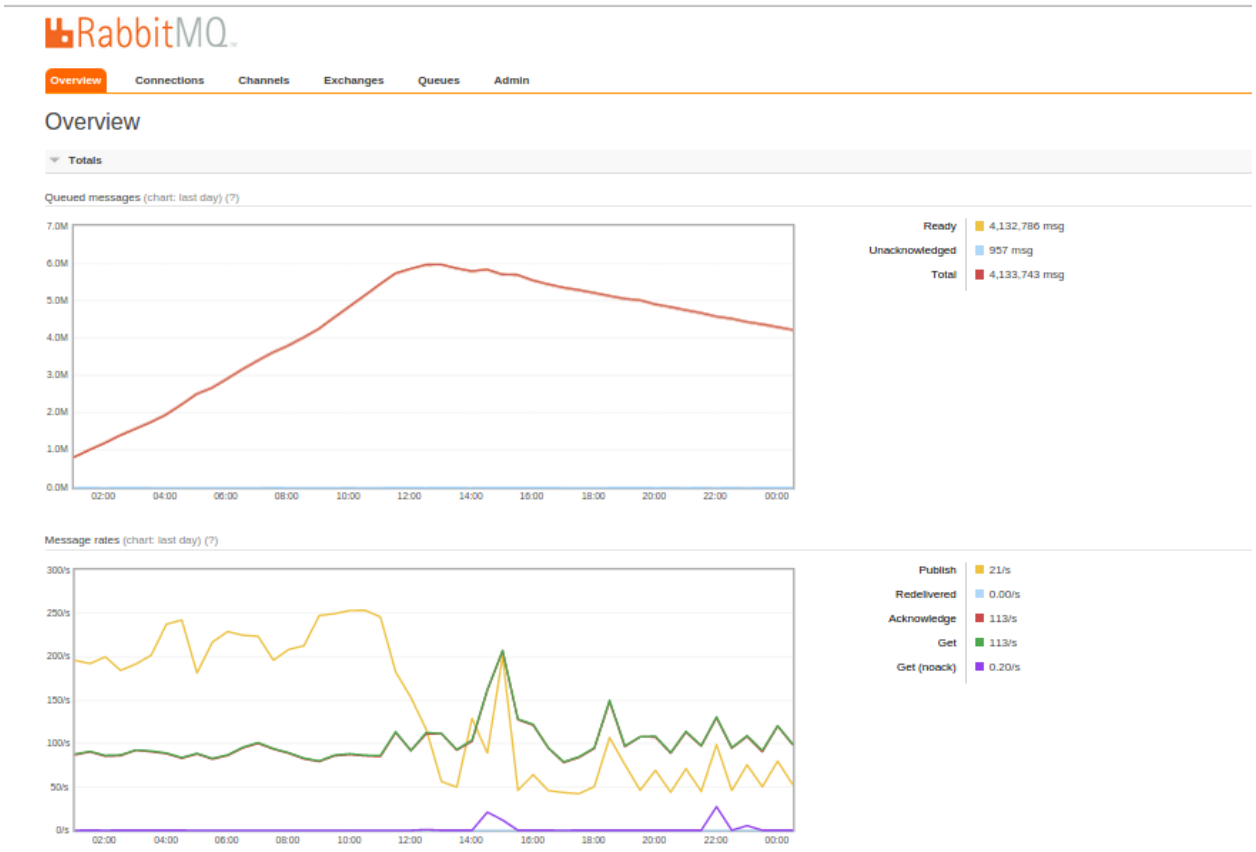
Overview					Messages			Message rates		
Name	Exclusive	Parameters	Policy	State	▲ Ready	Unacked	Total	incoming	deliver / get	ack
D0002199		D		running	4,059,997	512	4,060,509	0.00/s	142/s	142/s
D0001203		D		running	10,395	1,012	11,407	0.00/s	55/s	55/s
D0000350		D		idle	2,156	0	2,156	0.00/s	0.00/s	0.00/s
D0000714		D		idle	312	0	312	0.00/s		
D0000216		D		idle	125	0	125	0.00/s	0.00/s	0.00/s
D0000209		D		idle	96	0	96	1.8/s	0.00/s	0.00/s
D0000142		D		idle	60	0	60	6.4/s	0.00/s	0.00/s
D0002004		D		idle	42	0	42	0.00/s	0.00/s	0.00/s
D0000147		D		running	36	0	36	0.40/s	0.00/s	0.00/s
D0000164		D		running	31	0	31	0.60/s	0.00/s	0.00/s
D0000183		D		idle	23	0	23	0.00/s	0.00/s	0.00/s
D0000327		D		idle	22	0	22	0.00/s	0.00/s	0.00/s
D0000404		D		idle	20	0	20	0.00/s	0.00/s	0.00/s
D0000642		D		idle	19	0	19	0.00/s	0.00/s	0.00/s
D0000474		D		idle	16	0	16	0.00/s	0.00/s	0.00/s
D0001784		D		running	15	0	15	0.60/s	0.00/s	0.00/s
D0000389		D		idle	15	0	15	0.00/s	0.00/s	0.00/s
TRASH_NOTIFIED		D		idle	12	0	12	0.00/s		
D0000241		D		idle	12	0	12	0.00/s	0.00/s	0.00/s
D0001682		D		idle	11	0	11	0.00/s	0.00/s	0.00/s
D0001397		D		idle	11	0	11	2.2/s	0.00/s	0.00/s
D0000487		D		idle	11	0	11	0.00/s	0.00/s	0.00/s
D0001365		D		idle	9	0	9	0.00/s	0.00/s	0.00/s
D0002180		D		idle	8	0	8	0.00/s	0.00/s	0.00/s
D0001377		D		idle	8	0	8	0.00/s	0.00/s	0.00/s
D0000065		D		idle	8	0	8	1.6/s	0.00/s	0.00/s
D0001386		D		idle	7	0	7	0.00/s	0.00/s	0.00/s

Tabla de Colas del Rabbit

La aplicacion web del rabbit nos muestra todas las colas existente en el sistema, ademas de mostrarnos el número de mensajes que contienen.

Con esta pantalla podemos determinar si la creacion de colas esta funcionando o ver que colas estan siendo consumidas o no.

Ademas de esto rabbit nos brinda graficos en el cual podemos apreciar la velocidad de consumo y publicación a tenido el sistema. Con esto podemos determinar si el consumo o la publicación esta muy lenta por algún motivo.



Velocidad de Consumo y Publicación

En esta página vemos dos gráficas, una muestra el número de mensajes que hay y hubo en el sistema. Y el otro nos muestra tanto la velocidad de consumo, como la de publicación.

Con estas gráficas nos damos cuentas de la velocidad con la que se estan insertando los mensajes en el rabbit. Tanto las líneas de publicación, como las de consumo deben ser iguales o parecidas

7.3 Mejoras Posibles

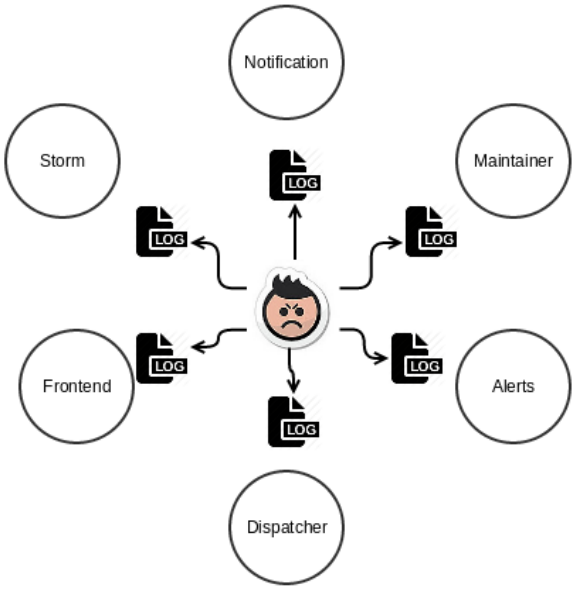
El sistema tiene un amplio rango de mejoras y una de ellas es la monitorización.

En un sistema distribuido, al tener modulos independientes, es necesario tener la mayor cantidad de herramientas posibles para detectar fallos.
Se ha pensado implementar las siguientes:

- Logs Centralizados
- Datos estadisticos

Logs Centralizados

Los modulos del sistema, al ser aplicaciones independientes, generan sus propios ficheros logs, donde registran todos y cada una de los eventos que realizan en todo momento.
Lo malo es que al tener distribuido toda esta información en diferentes máquina, se hace muy ineficiente poder rastrear el camino de un mensaje en concreto por todo el sistema.

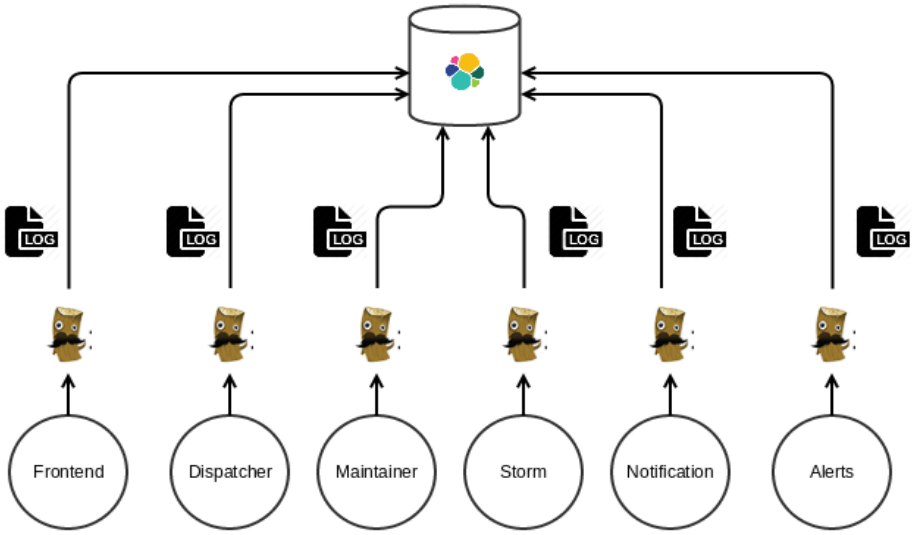


Modelo de logs Distribuidos

Una solución a esto es de almacenar toda esta información en un mismo sitio, así cada consulta de estos mensajes se harán siempre en un solo lugar.

Las herramientas que nos ayudarán a implementar esta funcionalidad son.

- Elasticsearch
- Logstash
- Kibana o Marvel



Modelo logs Centralizados

Metricas

Una vez tenemos el sistema de inserción trabajando en el entorno de producción, recibiendo datos de los diferentes clientes de la empresa, es imposible saber como se estan comportando los componentes o modulos del sistema ante trafico de datos en tiempo real.

De momento solo contamos con las interfaces web del rabbit ,el administrador del frontend o la interfaz web de storm para tener una visión de que esta pasando, pero no es suficiente.

Saber cuantos mensajes entran cada hora, cuántas colas se dan por minuto, cuántas alertas se han producido, toda esta información nos ayudará a:

- Ver si está funcionando cada parte del software
- Conocer el rendimiento del sistema en todo momento
- Detectar posibles fallos y mejoras

8.Pruebas de carga y rendimiento

Para determinar lo bien que responde los modulos a diferentes niveles de uso, se realizaron pruebas de carga sobre los modulos frontend y dispatcher para poder saber a partir de que número de llamadas concurrentes empiezan a ver fallos, como ser pérdidas de datos en el caso del frontend o lentitud en la entrega de colas en el caso del dispatcher.

8.1 Test Frontend

El frontend es la puerta de entrada al sistema de inserción, por eso es importante saber hasta que punto el módulo puede aguantar peticiones concurrentes.

Se realizaron pruebas de carga, para saber que tan rápido y fiable es el módulo, y a partir de que punto es necesario escalar para evitar la pérdida de datos.

Utilizando la herramienta locust hecha en python, se programaron escenarios para realizar los test y conseguir información acerca de que tan fiable es el módulo en situaciones extremas.

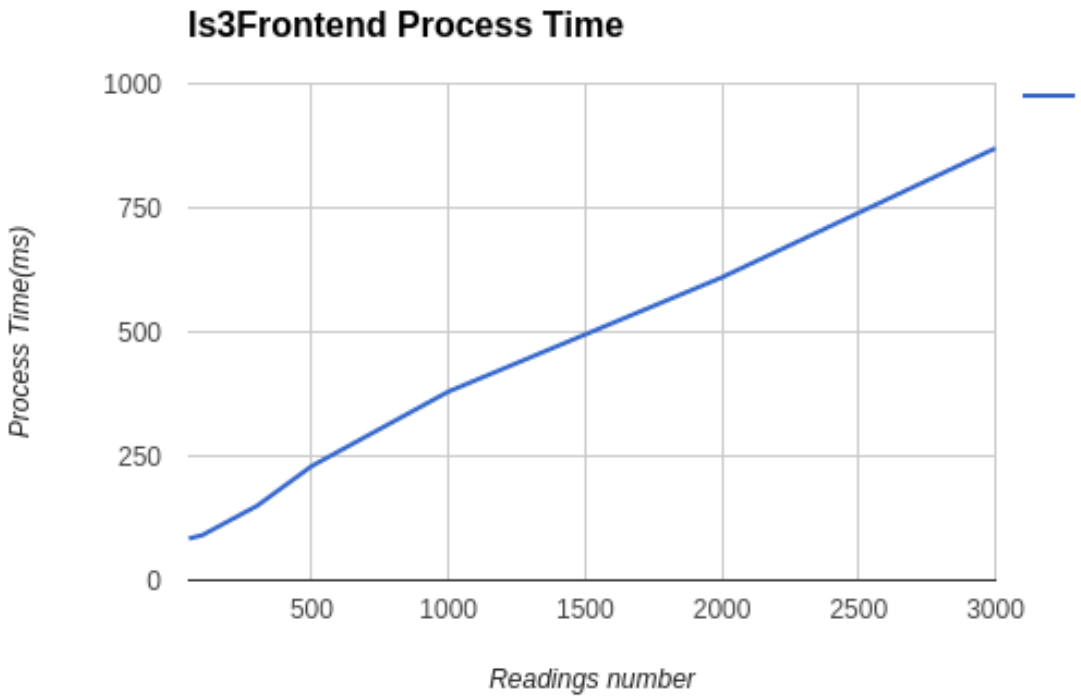
Se realizaron test al módulo antiguo y al nuevo y se sacaron los siguientes resultados

1.Test de carga Frontend nuevo

En este test se realizaron pruebas para saber el tiempo que toma en procesar una petición, según el número de mensajes que tenga para insertar.

Se ha usado el locust para simular un escenario, en el cual se realice una petición a la api frontend variando el número de lecturas que contiene la petición.

Los resultados son los siguientes:

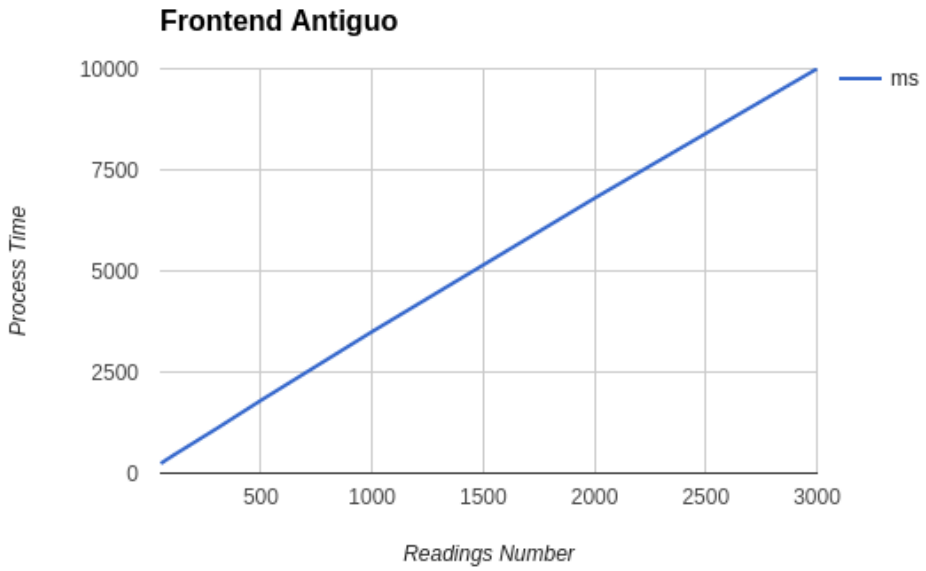


Tiempos que toman en procesar una llamada según el número de lecturas

Número de Mensajes	Tiempo de respuesta(ms)
50	84
100	91
300	150
500	230
1000	380
2000	610
3000	870

En las pruebas podemos ver que con 3000 lecturas introducidas en una misma llamada, se tarda menos de un segundo en ser insertados en el Rabbit.

Luego de poder ver velocidad de inserción del módulo, se comparó con el antiguo módulo frontend y los resultados fueron:



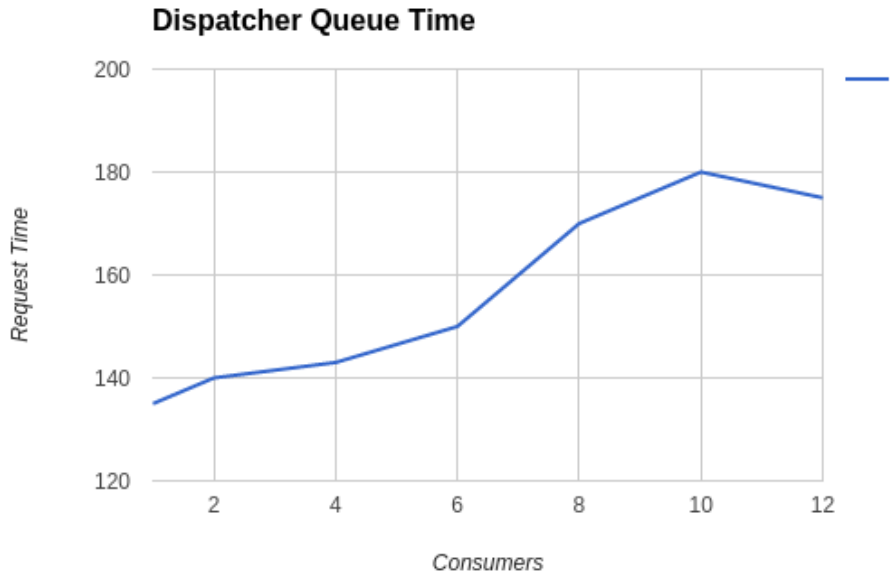
Número de Mensajes	Tiempo de respuesta(ms)
50	240
100	420
300	1100
500	1800
1000	3500
2000	6800
3000	10000

Como podemos obser los tiempos del nuevo módulo son mucho mejores que el anterior. Con esto demostramos que el nuevo módulo es más eficiente que el anterior.

8.2 Test Dispatcher

En este test se ha simulado la petición de colas y se ha calculado el tiempo medio que tardan en entregarlas, según el número de consumidores concurrentes.

Así con esta prueba podemos saber la velocidad del mongodb de lectura y escritura es la idónea para el sistema.



Tiempo de Peticiones al Dispatcher

Tabla Tiempo de respuesta Al pedir colas

Número de usuarios	Tiempo medio respuesta(ms)
1	130
2	140
4	145
6	157
8	170
10	180
12	180

En esta prueba se ha intentado ver dos cosas:

- Ver el tiempo medio que tarda en dar una cola
- Calcular el número de usuarios concurrentes puede soportar

En la realidad no se espera tener muchos consumidores de entrada pidiendo colas para trabajar, pero era necesario saber el rendimiento del dispatcher, con un número de consumidores razonable(1-12).

Y visto los resultados, los tiempos de entrega estan dentro de lo esperado para el funcionamiento del sistema en tiempo real.

9. Metodología y rigor

9.1 Scrum

Dado que el proyecto será llevado a cabo por un equipo de desarrolladores la metodología de trabajo elegida en este caso es SCRUM. Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible.

9.2 Seguimiento

Siguiendo el proceso de trabajo que nos indica la metodología Scrum se realizarán reuniones periódicas entre 7 o 10 días, según la carga de trabajo y horas que se planifique. En estas reuniones se determinarán con el director y encargado de la dirección del proyecto las diferentes tareas que se llevarán a cabo en el sprint(Nombre que recibe el tiempo de desarrollo de un grupo de tareas).

En esta reunión se realizará tres cosas:

Sprint review: Se hace una validación de todas las tareas programadas para el sprint anterior, y las que cumplen los expectativas esperadas pasan a estar completadas.

Sprint retrospection: El director del proyecto da su opinión acerca del sprint anterior, las tareas completadas y si se llego al objetivo esperado.

Sprint planning: Se hace la planificación de las tareas para el nuevo sprint asignándole un valor según el tiempo y esfuerzo que se cree tendrán estas. Las tareas que no se han podido completar del anterior sprint pasan a este nuevo.

9.3 Validación

Los test unitarios son importantes para comprobar el funcionamiento de cada una de las tareas asignadas en el sprint. Como norma general cada tarea debe tener sus test unitarios para considerarse finalizada.

10.Planificación Temporal

10.1 Tareas

El proyecto cumplira las siguientes fases de desarrollo:

1. Análisis de requerimiento
2. Diseño
3. Codificación
4. Integración y Pruebas
5. Bug Fixing

Análisis de Requerimientos

En esta tarea se analizó el sistema de inserción antiguo de la empresa, para conocer las funcionalidades y fallos que tiene.

A partir de aquí se empezó a diseñar el nuevo sistema.

Diseño del Sistema

Se empezó a diseñar una arquitectura orientada a servicios, donde cada funcionalidad del anterior sistema, es un módulo independiente en el nuevo

Codificación

Empieza el desarrollo de cada uno de los modulos diseñados en el paso anterior hasta tener una primera versión .

Al finalizar el desarrollo, se realizaron test un unitarios de cada modulo antes de poder liberar la primera versión del código.

Integración y Pruebas

Luego de tener la primera versión de cada módulo, se realizó el despliegue en el en el entorno de preproducción.

Se realizaron las primeras pruebas del sistema en conjunto funcionando.

Bug Fixing

Todos los fallos que fueron detectados en las pruebas de integración, pasaron por el proceso de bug fixing, que será la reparación de la parte del código más la ejecución de los test unitarios antes de desplegar en el entorno .

10.2 Diagrama de Gantt

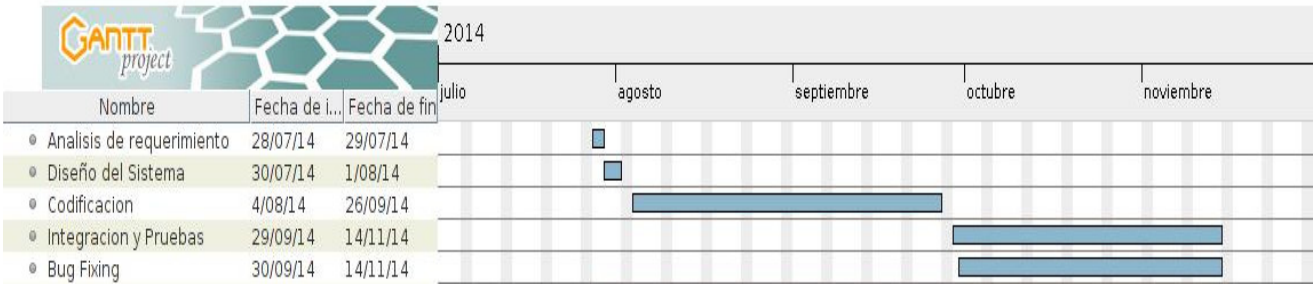


Diagrama de Gantt

11.1 Hardware

Dispositivo	Cantidad	Precio Unitario	Precio proporcional (5 meses)
Ordenador Portatil	1	900 €	125 €
Ordenador Torre	4	700 €	388 €
Maquinas Remotas	14	60 € al mes	4,200€
Total			4,713

Como se puede llegar a ver en la tabla, este sería el hardware que en primera instancia se usará para llevar a cabo el proyecto.

Los ordenadores torres son los que usará cada desarrollador en la oficina para poder desarrollar las tarea que se le asignan. Torres con 8 gb de ram y un procesador i7 de intel.

Las maquinas remotas son aquellas que se usara para desplegar tanto el entorno de producción como ser el entorno de preproducción.
Se calcula que se utilizaran 4 maquinas para el entorno de pre, como 10 maquinas para el entorno de pro. La diferencia es que el sistema distribuida como todo contará con alta disponibilidad, es decir tendrá replicación.
El aquiler de cada máquina individual tendrá un coste de 60 euros al mes, en general serian unos 1860 euros al mes. El proveedor de hosting que se usará hetzner.

11.2 Software

Software	Unidades	Precio	Precio Total(5 meses)
Ide Pycharm	2	0 €	0 €
Assembla	1	24 €/mes	120 €
MongoDB	3	0 €	0 €
Postgresql	2	0 €	0 €
Redis	2	0 €	0 €
SO Ubuntu 14.04	4	0 €	0 €
Nginx	5	0 €	0 €

En la parte de software no se realizarán muchos gastos ya que la mayoría de las herramientas que se utilizarán son opensource y son gratuitos.

Assembla sería la única herramienta que se contratará, alquilando una cuenta para poder gestionar los repositorios subversion que se tiene así como la planificación de las tareas mediante un sistema de tickets.
Se estima que el proyecto dure unos cuatro meses así que la cuenta costará unos 96 euros en total

11.3 Recursos Humanos

	Cantidad	Horas	Precio/Hora	Total
Director del proyecto	1	120 H	100 €	12000 €
Desarrolladores	4	600 H	60 €	144000 €
Totalº				156000 €

El equipo de trabajo contará con un director de proyecto que será el responsable de la direccion del proyecto. Se estima que el tiempo de horas que esté involucrado serán de 120 horas en total.

Luego el equipo de desarrollo contará contará con cuatro personas, tres desarrolladores y una persona de sistema.
El número de horas de cada persona será el máximo que dure el desarrollo del proyecto.

11.4 Coste de estructura

	Cantidad	**Consumo(kwh)**	**precio/Kwh**	**Total**
Consumo Energetico	2520h	0.302	0,18	136 €

* El coste de la luz se supone que será aproximadamente de 0,18€/KWh, Sacado de las tarifas de Septiembre 2014.

	Trabajadores	Superficie	Precio/m2	Total(5 meses)
Alquiler de espacio	5	50 m2	11 €/m2 y mes	2.750 €

11.5 Gastos imprevistos

Como costes imprevistos se considerarán un posible fallo en unos de los pc torres durante el periodo de realización del proyecto para el cual se estimará un coste de reparación de unos 200€. Asi como algun alquiler de una o mas maquina a hetzner que serian 60 Euros al mes durante unos 4 meses serian unos 240 Euros, seria un total de más unos 440 Euros

11.6 Coste Total

Puntos	**€**
Hardware	4713 €
Software	120 €
Recursos Humanos	156000 €
Coste de estructura	2886 €
Total	**163,719 €**

12. Conclusion

Se ha logrado, con éxito, el desarrollo de los diferentes modulos que forman parte del nuevo sistema de inserción de datos.

Cada uno de los modulos cumplen los objetivos planteados al principio del proyecto, que son los siguientes:

Modularidad, Sistema compuestos por modulos totalmente independientes y distribuidos en diferentes maquinas.

Mejora del rendimiento, visto los test de carga del frontend y dispatcher, los tiempos de ejecución de sus funcionalidades fueron óptimos. El nuevo frontend mostró tiempos menores a los del anterior.

Replicacion y escalabilidad, los modulos se han desarrollado para que sean totalmente replicables y escalables, todos los modulos son escalables a nivel aplicacion pero no a nivel de base de datos.

Detección de Fallos, la monitorización es una parte importante en todo sistema distribuido. Actualmente el sistema cuenta con la interfaz web para ver los mensajes que entran por el frontend, la web que nos brinda rabbitmq para ver las colas,los ratios de publicación y consumo, y los diferentes ficheros logs de cada modulo.
Pero estas herramientas no son suficientes para tener una visión total de los modulos del sistema, asi que se han planteado mejoras como ser la recolección de métricas y centralizar los mensajes logs de cada módulo en un solo lugar.

13.Bibliografia

[1] Apache Storm

https://storm.apache.org/

[2] RabbitMq

https://www.rabbitmq.com/

[3] Bottle Python

http://bottlepy.org/docs/dev/index.html

[4]Nginx

http://nginx.com/

[5]Uwsgi Tutorial

https://uwsgi-docs.readthedocs.org/en/latest/

[6] Ansible Tutorial

http://www.ansible.com/home

[7]Jenkins Official Doc

https://jenkins-ci.org/

[8] Pika Documentation

https://pika.readthedocs.org/en/0.9.14/

[9] Kombu Documentation

https://pypi.python.org/pypi/kombu

[10]Redis Db

http://redis.io/documentation

[11] MongoDb

https://www.mongodb.org/

[12]Scrum

http://www.desarrolloweb.com/manuales/metodologias-agil-desarrollo-software.html

[13] Computación distribuida

http://www.sc.ehu.es/acwlaalm/sdi/introduccion-slides.pdf